

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



杨帆 王钧玉 孙更新 编著

设计模式 从入门到精通



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

<http://www.phei.com.cn>

内容简介

设计模式从入门到精通

杨帆 王钧玉 孙更新 编著

2012.6.16

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书使用Java语言来描述经典的GoF的23种设计模式，在讲解过程中涉及了JDK 6.0中的新特性，全书采用案例驱动的形式，以一个完整的超市系统案例统领了全部知识点。本书以案例项目工程为主线，以应用为目的，循序渐进地讲解了设计模式的具体应用方法，易学易用，并且结合案例驱动形式，可以使读者将各种设计模式真正运用到实际开发中，避免理论与实践脱节的问题。

本书适用于对设计模式不甚了解的初学者，同时也适合具有一定编程基础、需要提高实践技术的程序员作为参考用书。本书还可作为高等院校计算机等专业及相关培训学校的指导教材。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目(CIP)数据

设计模式从入门到精通/杨帆，王钧玉，孙更新编著. —北京：电子工业出版社，2010.8
ISBN 978-7-121-11560-8

I. ①设… II. ①杨…②王…③孙… III. ①Java语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2010)第154726号

责任编辑：李红玉

文字编辑：易 昆

印 刷：北京天竺颖华印刷厂

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

北京市海淀区翠微东里甲2号 邮编：100036

开 本：787×1092 1/16 印张：33.25 字数：851.2千字

印 次：2010年8月第1次印刷

定 价：62.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至zlt@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：(010) 88258888。

前言

自GoF推出《设计模式》这本经典书籍以来，在软件设计界，学习使用模式的风潮就没有停止过，设计模式是前人对于软件设计经验的总结，非常具有学习的价值。

本书的作者具有丰富的实际开发经验和培训经验，鉴于多年的培训工作和软件开发经历，他们能很好地把握初学者对于模式的学习需求。本书是设计模式的实用性入门和进阶书籍，内容安排上注重实用，可以使初学者迅速学以致用。本书非常适合熟悉Java编程但是对设计模式经验相对较少的读者阅读。

本书采用案例驱动的形式，用一套完整的超市系统统领全书。书中第1章~第3章介绍了面向对象的设计方法以及设计模式的起源和优点，讲解了UML的发展历史以及常见的关系图。

第4章~第8章详细讲解了创建型模式的各个模式，包括了超市中各种需要创建的对象，解决了商品上架、捆绑销售、连锁店加盟、总店地位等问题。

第9章~第15章详细讲解了结构型模式的各个模式，解决的问题主要有：电源插座问题、游戏机配件问题、手机展示问题、宣传海报设计问题等。在案例中还穿插使用了创建型模式以加深读者对两大类型的模式的理解。

第16章~第26章详细讲解了行为型模式的各个模式，解决的主要问题有：连锁店客服专线问题、超市促销宣传资料的问题、商品进货审批中的问题、设计灵活高效的收银程序问题、超市在不同时段的运营状态问题、商品打折问题、设计不同糕点的制作流程问题等。在示例中综合运用了创建型模式、结构型模式、行为型模式，可以使读者对不同模式之间的合作加深印象。

第27章对所有模式进行了总结并提出了对未来模式的学习方向。

总体来说，本书具有以下特点：

- 内容全面，实例丰富，讲解循序渐进。
- 以基础知识紧密结合典型实例，具有很强的操作性和实用性。
- 从实际应用的角度出发，可以帮助读者以最快的速度学会使用Java语言开发设计模式，全面提高程序员的开发技术水平和面向对象的设计能力。

- 采用案例驱动模式，不仅能使读者掌握好知识，更能使其在实际开发中学以致用。
- 整体案例中贯穿了全部章节的知识，读者可以以此更深刻地理解模式的相互合作编程法。

本书在编写过程中，得到了很多专家和同事的大力支持，在此一并表示感谢。同时由于作者水平有限，加上时间仓促，书中难免存在疏漏和不妥之处，欢迎广大读者批评指正。

图系关的

为方便读者阅读，若需要本书配套资料，请登录“北京美迪亚电子信息有限公司”(<http://www.medias.com.cn>)，在“资料下载”页面进行下载。

目 录

第1章 设计模式初见	1
1.1 一切从某个小超市开始	1
1.2 为何使用设计模式	1
1.3 设计模式分类	2
1.4 阅读建议与学习资源	3
第2章 面向对象设计原则	4
2.1 软件的维护代价	4
2.2 面向对象设计原则	4
2.2.1 基础原则：“开一闭”原则 (OCP)	4
2.2.2 单一职责原则 (SRP)	7
2.2.3 里氏替换原则 (LSP)	10
2.2.4 依赖倒置原则 (DIP)	12
2.2.5 接口隔离原则 (ISP)	13
2.3 Java面向对象的支持	14
第3章 统一建模语言UML概述	15
3.1 UML发展史	15
3.2 UML中的关系	17
3.2.1 依赖关系	17
3.2.2 继承关系	17
3.2.3 实现关系	17
3.2.4 关联关系	18
3.2.5 聚合关系	18
3.2.6 组合关系	18
3.3 UML中的图形类型	19
3.4 UML工具软件	19
第4章 工厂方法模式 (Factory Method)	21
4.1 商品上架遇到的问题	21
4.2 工厂方法模式的结构	24
4.2.1 简单工厂模式	24
4.2.2 工厂方法模式	29
4.2.3 使用工厂方法模式解决商品上 架问题	33
4.2.4 工厂模式在JDK中的实例	42
4.2.5 工厂方法模式的使用范围	44
4.2.6 简单工厂与其他模式的区别	44
4.3 工厂方法模式总结	44
第5章 抽象工厂模式 (Abstract Factory)	45
5.1 深入探讨商品上架遇到的问题	46
5.2 抽象工厂模式的结构	51
5.2.1 抽象工厂模式	51
5.2.2 使用抽象工厂模式解决商品上 架问题	58
5.2.3 抽象工厂模式在JDK中的实例	66
5.2.4 抽象工厂模式的使用范围	71
5.2.5 与其他模式的关系	71
5.3 抽象工厂模式总结	72
第6章 建造者模式 (Builder Factory)	73
6.1 商品捆绑销售的问题	73
6.2 建造者模式的结构	76
6.2.1 建造者模式	76
6.2.2 使用建造者模式解决捆绑销售 问题	83
6.2.3 建造者模式的实际使用案例	92
6.2.4 建造者模式的使用范围	92
6.2.5 与其他模式的关系	93
6.3 建造者模式总结	93
第7章 原型模式 (Prototype)	94
7.1 新连锁店开张	94
7.2 原型模式的结构	97
7.2.1 原型模式	97
7.2.2 浅克隆与深克隆	102
7.2.3 使用原型模式解决连锁店 问题	109
7.2.4 原型模式在JDK中的应用实 例	116
7.2.5 原型模式的使用范围	116
7.2.6 与其他模式的关系	117
7.3 原型模式总结	117

第8章 单例模式 (Singleton)	118	11.2.1 代理模式	173
8.1 连锁店总店的唯一性问题	118	11.2.2 使用代理模式解决手机展示过程中的问题	177
8.2 单例模式的结构	120	11.2.3 几种不同的代理模式	186
8.2.1 单例模式	120	11.2.4 代理模式的使用范围	192
8.2.2 多线程情况下的单例模式	122	11.2.5 与其他模式的关系	193
8.2.3 单例模式的双重检查模式		11.3 代理模式总结	193
DCL	126	第12章 外观模式 (Facade)	194
8.2.4 使用单例模式解决连锁店总店的问题	129	12.1 顾客的意见——购买大件商品时手续繁多	194
8.2.5 单例模式在JDK中的实例	137	12.2 外观模式的结构	195
8.2.6 单例模式的使用范围	140	12.2.1 外观模式	195
8.2.7 与其他模式的关系	140	12.2.2 使用外观模式解决顾客的一站式服务	201
8.3 单例模式总结	141	12.2.3 外观模式在JDK中的实例	206
第9章 适配器模式 (Adapter)	142	12.2.4 外观模式的使用范围及优点	209
9.1 家电区的麻烦	142	12.2.5 与其他模式的关系	210
9.2 适配器的结构	143	12.3 外观模式总结	210
9.2.1 类适配器	143	第13章 装饰模式 (Decorator)	212
9.2.2 对象适配器	144	13.1 如何解决销售人员的能力固化问题	212
9.2.3 两种适配器的对比	145	13.2 装饰模式的结构	213
9.2.4 用适配器模式解决电源插头问题	146	13.2.1 装饰模式	213
9.2.5 双向适配器	149	13.2.2 使用装饰模式解决销售人员问题	217
9.2.6 适配器模式在JDK中的实例	150	13.2.3 装饰模式在JDK中的实例	223
9.2.7 适配器的适用范围	155	13.2.4 装饰模式的使用范围	224
9.2.8 与其他模式的关系	155	13.2.5 与其他模式的关系	225
9.3 适配器模式总结	156	13.3 装饰模式总结	226
第10章 桥接模式 (Bridge)	157	第14章 组合模式 (Composite)	227
10.1 游戏机销售专柜引发的思考	157	14.1 如何描述超市的组织结构	227
10.2 桥接模式的结构	158	14.2 组合模式的结构	229
10.2.1 桥接模式	158	14.2.1 组合模式	229
10.2.2 使用桥接模式解决游戏机销售的问题	163	14.2.2 使用桥接模式解决超市组织结构问题	236
10.2.3 桥接模式在JDK中的实例	168	14.2.3 组合模式在JDK中的实例	244
10.2.4 桥接模式的使用范围	171	14.2.4 组合模式的使用范围	248
10.2.5 与其他模式的关系	171	14.2.5 与其他模式的关系	248
10.3 桥接模式总结	171		
第12章 代理模式 (Proxy)	172		
11.1 手机柜台遇到的问题	172		
11.2 代理模式的结构	173		

14.3 组合模式总结	248	18.2.3 用责任链模式建立明晰的进 货审批流程	332
第15章 享元模式 (Flyweight)	249	18.2.4 责任链模式在JDK中的实例	338
15.1 宣传海报设计过程中的思考	249	18.2.5 责任链模式的使用范围	342
15.2 享元模式的结构	250	18.2.6 与其他模式的关系	343
15.2.1 单纯享元模式	250	18.3 责任链模式总结	343
15.2.2 复合享元模式	255	第19章 迭代器模式 (Iterator)	344
15.2.3 使用享元模式解决宣传海报 的设计问题	260	19.1 收银员的商品处理效率亟待提高	344
15.2.4 享元模式在JDK中的实例	272	19.2 迭代器模式的结构	350
15.2.5 享元模式的使用范围	274	19.2.1 迭代器模式	350
15.2.6 与其他模式的关系	274	19.2.2 用迭代器模式统一处理各类 商品	354
15.3 享元模式总结	275	19.2.3 迭代器模式在JDK中的实例	362
第16章 命令模式 (Command)	276	19.2.4 迭代器模式的使用范围及优 点	365
16.1 连锁店客服专线的问题	276	19.2.5 与其他模式的关系	365
16.2 命令模式的结构	279	19.3 外观模式总结	365
16.2.1 命令模式	279	第20章 访问者模式 (Visitor)	366
16.2.2 使用命令模式解决客服电话 的问题	289	20.1 再谈收银员的效率问题	366
16.2.3 命令模式在JDK中的实例	297	20.2 访问者模式的结构	369
16.2.4 命令模式的使用范围	298	20.2.1 静态、动态、单分派、多分 派、双重分派	369
16.2.5 与其他模式的关系	299	20.2.2 访问者模式	374
16.3 命令模式总结	299	20.2.3 用访问者模式设计灵活高效 的收银程序	381
第17章 观察者模式 (Observer)	300	20.2.4 访问者模式在JDK中的实例	387
17.1 连锁店宣传资料的发放问题	300	20.2.5 访问者模式的使用范围及优 点	388
17.2 观察者模式的结构	304	20.2.6 与其他模式的关系	389
17.2.1 观察者模式	305	20.3 访问者模式总结	389
17.2.2 使用观察者模式构建发布/订 阅模型	309	第21章 状态模式 (State)	390
17.2.3 观察者模式在JDK中的实 例	316	21.1 如何调整超市的运营状态	390
17.2.4 观察者模式的使用范围	321	21.2 状态模式的结构	393
17.2.5 与其他模式的关系	322	21.2.1 状态模式	393
17.3 观察者模式总结	323	21.2.2 用状态模式设计超市在不同 时段的打折状态	403
第18章 责任链模式 (Chain of Responsibility)	324	21.2.3 状态模式的使用范围及优点	408
18.1 商品进货审批中的问题	324	21.2.4 与其他模式的关系	408
18.2 责任链模式的结构	328	21.3 状态模式总结	408
18.2.1 责任链模式	328		
18.2.2 纯的和不承担的责任链模式	332		

第22章 备忘录模式 (Memento)	409	第25章 模板方法模式 (Template Method)	483
22.1 服务器硬盘坏了, 营业数据全丢了	409	25.1 超市面包房的糕点制作流程规范化问题	483
22.2 备忘录模式的结构	411	25.2 模板方法模式的结构	487
22.2.1 备忘录模式白箱实现	412	25.2.1 模板方法模式	487
22.2.2 备忘录模式黑箱实现	415	25.2.2 用模板方法模式设计不同糕点的制作流程	491
22.2.3 用备忘录模式设计可靠的数 据保存系统	421	25.2.3 模板方法模式在JDK中的实 例	497
22.2.4 备忘录模式在JDK中的实例	431	25.2.4 模板方法模式的使用范围及 优点	499
22.2.5 备忘录模式的特点	432	25.2.5 与其他模式的关系	500
22.2.6 与其他模式的关系	433	25.3 模板方法模式总结	500
22.3 备忘录模式总结	433	第26章 解释器模式 (Interpreter)	501
第23章 策略模式 (Strategy)	434	26.1 新店开在了外国人聚居区	501
23.1 最常用的促销手段——打折	434	26.2 解释器模式的结构	504
23.2 策略模式的结构	437	26.2.1 解释器模式	504
23.2.1 策略模式	437	26.2.2 用解释器模式设计超市内的 多语言指示牌	510
23.2.2 用策略模式构建灵活的打折 方式	447	26.2.3 解释器模式在JDK中的实例	516
23.2.3 策略模式在JDK中的实例	459	26.2.4 解释器模式的使用范围及优 点	517
23.2.4 策略模式的使用范围及优点	460	26.2.5 与其他模式的关系	517
23.2.5 与其他模式的关系	461	26.3 解释器模式总结	518
23.3 策略模式总结	461	第27章 设计模式总结	519
第24章 调停者模式 (Mediator)	462	27.1 回顾设计模式在超市发展中的足 迹	519
24.1 超市经营的核心——库存管理	462	27.2 各模式之间的关系及演变图	521
24.2 调停者模式的结构	469	27.3 新的知识新的起点	522
24.2.1 调停者模式	469	27.3.1 Java EE设计模式	522
24.2.2 用调停者模式协调进货/销售/ 盘点之间的关系	475	27.3.2 架构风格模式	523
24.2.3 调停者模式在JDK中的实例	481	27.3.3 架构模式	523
24.2.4 调停者模式的适用范围及优 缺点	482	27.3.4 分层架构模式	524
24.2.5 与其他模式的关系	482	27.4 学习建议	524
24.3 调停者模式总结	482		

第1章 设计模式初见

设计模式（Design Pattern）是一套经过分类的、被反复使用的软件代码设计经验的总结。使用设计模式是为了可复用代码，让代码更容易被理解，保证代码的可靠性。通常来说，设计模式是软件复用的基础理论，它使代码编制真正工程化。

设计模式最初是在建筑学中被提出的，建筑师克里斯托佛·亚历山大在1970年代编撰了一本汇集设计模式的书，但是设计模式的思想在建筑设计领域里的影响远没有后来在软件开发领域里传播得广泛和深远。

软件设计中的设计模式是在GoF（“四人帮”，指Gamma、Helm、Johnson & Vlissides、Addison-Wesley四人）合著的《设计模式》一书中第一次提出的，随后被规范化。本书提出的23种基本设计模式便属于《设计模式》中所提及的经典的模式。

1.1 一切从某个小超市开始

在软件工程领域中研究一种具体的技术，通常都会借助一个具体的案例来分析和学习，在本书中也不例外，在每一章节的学习过程中，读者除了要学习和分析模式的案例，还将学习如何使用设计模式来解决一个现实工程中存在的问题。

各章节的案例都来源于一个“超市”案例，因为“超市”对于大众来说都比较熟悉，其中发生的问题也比较容易理解。

软件设计中的超市是什么样子的呢？其实本软件设计中就是使用软件来模拟人经营一个超市，现实生活中的超市中发生的各种情况在软件环境中都要提及并加以处理。比如要进行商品的上架、仓库进货、打折销售、客户服务、广告宣传等。

在学习每一个模式时，为了达到良好的学习效果，读者最好能了解一下每一章涉及的超市问题发生的原因及需求，这对理解模式的意图是十分关键的。

1.2 为何使用设计模式

要回答为何要使用设计模式这个问题，必须要知道设计模式的优点。设计模式的优点如下：

复用解决方案

在代码设计中通常会遇到需要设计的方案和以前设计的某个方案类似的问题，比如之前已经设计过一个论坛系统，现在又要设计一个讨论版系统。这时，比较好的解决方案就是最大限度地利用之前设计的代码。

设计模式的主要思想就是“复用”，通过复用已经确认的设计，能够在解决问题的过程中使用最小的成本获得最大的效益，而且可以在学习他人经验的过程中获利，不用再为那些总是会重复出现的问题重复设计解决方案。

设计模式将设计方法标准化

开发团队中的交流和协作通常都要使用共同的词汇和要对问题达成共识。设计模式在项目

的分析和设计阶段提供了共同的基准点。如果项目团队中的所有成员都熟悉设计模式就可以产生很高的沟通效率，比如团队成员A说，我认为这个地方适合使用“策略模式”来实现算法改变。团队成员B说，我不同意你的观点，我认为这个地方的算法改变是跟对象状态相关的，应该使用“状态模式”。这样表述起来非常简明扼要。

设计模式可以提高个人和团队的设计能力

在开发团队中使用设计模式的经验证明，设计模式既可以帮助开发人员个人提高个人水平，也可以帮助团队提高整体实力。这是因为，经验少的团队成员亲眼看到已经掌握设计模式的资深开发人员如何快速从中获益后，他们会更加自发、主动地去学习这些强大的知识。

设计模式使软件更容易修改和维护

设计模式能够使软件更容易修改和维护的原因在于：这些模式都是久经考验的解决方案。所以，它们的结构都是长期发展形成的，比新构思的解决方案更善于应对变化。而且，这些设计模式所用的代码往往更易于理解，从而使代码更易于维护。使用设计模式对开发人员、维护人员、测试人员都有好处，所以使用设计模式是一种“多赢”的设计方法。

如果说学习某种具体的编程语言是练习程序设计的“招式”的话，那么学习设计模式无疑就是练习程序设计的“内功”，而“内功”通常才是一个高手制胜的关键。

1.3 设计模式分类

经典的GoF的23种设计模式分为三个类别。

- 创建型模式：用于创建对象。对象的创建会消耗掉系统的很多资源，所以单独对对象的创建进行研究，以便能够高效地创建对象就是创建型模式要探讨的问题。
- 结构型模式：用于构建类间的关系。如何设计对象的结构、继承和依赖关系会影响到后续程序的维护性，代码的健壮性、耦合性等。这些因素需要使用结构型模式来优化。
- 行为型模式：用于控制对象的行为。如果对象的行为设计得好，那么对象的行为就会更清晰，它们之间的协作效率就会更高。

每一类设计模式都有多种具体模式，如表1-1所示。

表1-1 设计模式分类

模式分类	模式名称
创建型模式	工厂方法模式
	抽象工厂模式
	建造者模式
	原型模式
	单例模式
结构型模式	适配器模式
	桥接模式
	代理模式
	外观模式
	装饰模式
	组合模式
	享元模式

(续表)

模式分类	模式名称
行为型模式	命令模式
	观察者模式
	责任链模式
	迭代器模式
	访问者模式
	状态模式
	备忘录模式
	策略模式
	调停者模式
	模板方法模式
	解释器模式

1.4 阅读建议与学习资源

- 下面推荐的是公认的设计模式经典书籍：
- 《设计模式：可复用面向对象软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*）；
 - 《设计模式精解》（*Design Patterns Explained*）；
 - 《重构——改善既有代码的设计》（*Refactoring: Improving the Design of Existing Code*）；
 - 《软件复用技术：在系统开发过程中考虑复用》（*Software Reuse Techniques Adding Reuse to the Systems Development Process*）；
 - 《软件复用：结构、过程和组织》（*Software Reuse Architecture, Process and Organization for Business Success*）；
 - 《企业应用架构模式》（*Patterns of Enterprise Application Architecture*）；
 - 《UML面向对象设计基础》（*Fundamentals of Object-Oriented Design in UML*）；
 - 《敏捷软件开发——原则、模式与实践》（*Agile Software Development: Principles, Patterns, and Practices*）；
 - 《软件工程——实践者的研究方法（原书第5版）》（*Software Engineering: A Practitioner's Approach, Fifth Edition*）；
 - 《计算机程序设计艺术》（*The Art of Computer Programming*）。
- 学习设计模式的经典网站资源：
- 维基百科：<http://en.wikipedia.org>或<http://zh.wikipedia.org>。
 - UML软件工程组织：<http://www.uml.org.cn/>。
 - J道：<http://www.jdon.com/>。
 - CSDN综合社区：<http://www.csdn.net>。
 - WeLie模式频道：<http://www.welie.com/patterns/>。

第2章 面向对象设计原则

毫无疑问，当前最流行的软件设计方式就是面向对象的设计方式。在程序设计的过程中，使用面向对象的设计方式会给大规模的软件开发带来清新的活力，提高编程的效率，但是如果对面向对象的设计方法掌握不好也会带来很多的问题，本章就介绍一下面向对象的设计原则，这也是设计模式需要遵守的原则。

2.1 软件的维护代价

软件设计通常来说具有长期性，设计出软件之后还需要根据需求的变更对其进行维护。维护通常分为几类：功能扩充、功能修改、功能删除。设计不当通常会带来如下问题：

- 僵化性（Rigidity）：设计难以改变。具有这种特性的软件将无法维护。
- 脆弱性（Fragility）：设计易于遭到破坏。这种特性使得其他程序员在使用设计好的类时有可能将其破坏。
- 牢固性（Immobility）：设计难以重用。无法重用的软件模块将会导致大量不必要的劳动。
- 不必要的复杂性（Needless Complexity）：过分设计导致设计复杂，影响开发效率和性能。
- 不必要的重复（Needless Repetition）：过多的重复违反了不要重复自己的DRY（don't repeat yourself）原则。
- 晦涩性（Opacity）：混乱的表达导致程序不可读。

如果设计的应用程序具备以上问题，很显然，维护的成本将是很高的，但是如果采用正确的面向对象设计原则，将会解决这些问题。

2.2 面向对象设计原则

良好的面向对象设计方法需要遵循一些基本原则，如单一职责原则（SRP）、开一闭原则（OCP）、里氏替换原则（LSP）、依赖倒置原则（DIP）、接口隔离原则（ISP）等。下面依次进行举例介绍。

2.2.1 基础原则：“开一闭”原则（OCP）

“开一闭”原则的含义是：一个软件实体应当对扩展开放，对修改关闭。

在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。从另外一个角度讲，就是所谓的“对可变性封装原则”。“对可变性封装原则”意味着两点：

- 一种可变性不应当分散于很多代码片段中，而应当被封装到一个对象里面。同一种可变性的不同表象意味着同一个继承等级结构中的具体子类。
- 一种可变性不应当与另一种可变性混合在一起。类的设计应该具备特定的可变性而不是众多的可变性。

下面的例子将说明此原则的意义，假设要用代码描述一个汽车类Car，其中要定义汽车名称和发动机类型，代码如下：

代码片段1 Car.java

```

1 package cn.steven.oo.ocp;
2
3 /**
4  * 违反开一闭原则的汽车类
5  */
6 public class Car {
7
8     /**
9      * 汽车名称
10     */
11     private String name;
12
13     /**
14      * 汽车引擎
15     */
16     private String eng;
17
18     public String getName() {
19         return name;
20     }
21
22     public void setName(String name) {
23         this.name = name;
24     }
25
26     public String getEng() {
27         return eng;
28     }
29
30     public void setEng(String eng) {
31         this.eng = eng;
32     }
33
34     public void print() {
35         System.out.println("引擎是" + eng + "的" + name);
36     }
37
38     public static void main(String[] args) {
39         Car car = new Car();
40         car.setEng("V4发动机");
41         car.setName("皮卡");
42         car.print();
43         //运行结果
44         //引擎是V4发动机的皮卡
45     }

```

代码片段3 Document.java

```

1 package cn.steven.oo.srp;

```


46

47 }

可见,虽然这个类的使用是没有问题的,但是要更换其中的发动机类型将变得很复杂(上面例子中使用的是字符串,效果不明显,假设发动机是一个单独的类这个问题就明显了),所以发动机类型的这种变化应该被独立设定出来。

修改后的代码如下:

代码片段2 CarModi.java

```
1 package cn.steven.oo.ocp;
2
3 /**
4  * 符合开一闭原则的汽车类
5  */
6 public class CarModi {
7
8     /**
9      * 汽车名称
10     */
11     private String name;
12
13     /**
14      * 汽车引擎
15     */
16     private IEng eng;
17
18     public String getName() {
19         return name;
20     }
21
22     public void setName(String name) {
23         this.name = name;
24     }
25
26     public IEng getEng() {
27         return eng;
28     }
29
30     public void setEng(IEng eng) {
31         this.eng = eng;
32     }
33
34     public void print() {
35         System.out.println("引擎是" + eng.getEng() + "的"
36             + name);
37     }
38
39     public static void main(String[] args) {
40         CarModi car = new CarModi();
41         car.setEng(new V4Eng());
```

```

42     car.setName("皮卡");
43     car.print();
44     car.setEng(new V8Eng());
45     car.print();
46     // 运行结果
47     // 引擎是V4的皮卡
48     // 引擎是V8的皮卡
49 }
50
51 }
52
53 /**
54  * 引擎接口
55  */
56 interface IEng {
57     public String getEng();
58 }
59
60 /**
61  * V4引擎
62  */
63 class V4Eng implements IEng {
64     private String eng = "V4";
65
66     public String getEng() {
67         return eng;
68     }
69 }
70
71 /**
72  * V8引擎
73  */
74 class V8Eng implements IEng {
75     private String eng = "V8";
76
77     public String getEng() {
78         return eng;
79     }
80 }

```

由代码可见，现在已经可以自由地更换引擎而不用修改汽车对象本身了。

2.2.2 单一职责原则 (SRP)

单一职责原则的含义是：就一个类而言，应该仅有一个引起它变化的原因。

在构造对象时，应将对象的不同职责分离至多个类中，从而确保引起该类变化的原因只有一个。使用此原则可以提高内聚，降低耦合度。

比如有一个文档类，这个类目前的职责是负责维护文档内容和打印，代码如下：

代码片段3 Document.java

```
1 package cn.steven.oo.srp;
```



```

2
3 /**
4  * 违反SRP的文档类
5  */
6 public class Document {
7
8     /**
9     * 文档内容
10    */
11    private String content;
12
13    public String getContent() {
14        return content;
15    }
16
17    public void setContent(String content) {
18        this.content = content;
19    }
20
21    /**
22    * 打印方法
23    */
24    public void printMethod() {
25        System.out.println("将文档内容打印至控制台");
26    }
27 }

```

由代码可见，**Document**类有两种使其改变的因素，一个是文档内容的改变，这个是此类的本分职责；另一个是，如果现在需要将文档传送至文件或者网络进行打印，那么势必要修改**Document**类中的print方法。这样一来，**Document**类就有了两种使其改变的因素，违反了SRP原则。修改方法就是将print方法放在另一个职责中：

代码片段4 DocumentModi.java

```

1 package cn.steven.oo.srp;
2
3 /**
4  * 符合单一职责原则的文档类
5  */
6 public class DocumentModi {
7
8     /**
9     * 文档内容
10    */
11    private String content;
12
13    public String getContent() {
14        return content;
15    }
16
17    public void setContent(String content) {

```

```

18     this.content = content;
19 }
20
21 /**
22  * 打印对象
23  */
24 private IPrint print;
25
26 public IPrint getPrint() {
27     return print;
28 }
29
30 public void setPrint(IPrint print) {
31     this.print = print;
32 }
33
34 /**
35  * 打印方法
36  */
37 public void printMethod() {
38     print.print();
39 }
40 }
41
42 /**
43  * 打印接口
44  */
45 interface IPrint {
46     public void print();
47 }
48
49 /**
50  * 控制台打印
51  */
52 class ConsolePrint implements IPrint {
53
54     @Override
55     public void print() {
56         System.out.println("将文档内容打印至控制台");
57     }
58 }
59
60
61 /**
62  * 文件打印
63  */
64 class FilePrint implements IPrint {
65
66     @Override
67     public void print() {
68         System.out.println("将文档内容输出至文件");

```

```

69     }
70
71 }

```

由代码可见，经过修改后的代码中Document和IPrint类的职责都很单一，这样耦合度就会降低，从而带来可维护性的好处。

2.2.3 里氏替换原则（LSP）

里氏替换原则的含义是：若对每个类S的对象O1，都存在一个类T的对象O2，使得在所有针对T编写的程序P中，用O1替换O2后，程序P行为功能不变，则S是T的子类。

该原则的具体应用体现在继承关系上，在实现继承时，子类必须能替换掉它们的基类。如果一个软件代码中使用的是基类的话那么也一定可以使用其子类，但反过来的代换则可以不成立。

下面的例子解决了一个问题：为什么正方形不可以设计为矩形的子类：

代码片段5 Rectangle.java

```

1  package cn.steven.oo.lsp;
2
3  /**
4   * 违反里氏替换原则的继承
5   */
6  public class Rectangle {
7      /**
8       * 长方形可以单独设计宽和高
9       */
10     private int width;
11     private int height;
12
13     public int getWidth() {
14         return width;
15     }
16
17     public void setWidth(int width) {
18         this.width = width;
19     }
20
21     public int getHeight() {
22         return height;
23     }
24
25     public void setHeight(int height) {
26         this.height = height;
27     }
28 }
29
30 class Square extends Rectangle{
31
32     /**
33      * 正方形的宽和高等长所以将对高的操作全部转移至宽中
34     */

```

```
35 public int getHeight() {
36     return getWidth();
37 }
38
39 public void setHeight(int height) {
40     this.setWidth(height);
41 }
42 }
```

代码中将正方形设计为矩形的子类，但是实际上正方形的特点和矩形不一样，因为它的宽和高不能独立变化，所以就会在编码过程中出现问题。修改后的代码如下：

代码片段6 RectangleModi.java

```
1 package cn.steven.oo.lsp;
2
3 /**
4  * 符合里氏替换原则的继承
5  */
6 public class RectangleModi {
7     /**
8      * 长方形可以单独设计宽和高
9      */
10    private int width;
11    private int height;
12
13    public int getWidth() {
14        return width;
15    }
16
17    public void setWidth(int width) {
18        this.width = width;
19    }
20
21    public int getHeight() {
22        return height;
23    }
24
25    public void setHeight(int height) {
26        this.height = height;
27    }
28 }
29
30 class SquareModi {
31     /**
32      * 正方形的宽和高等长
33      */
34    private int length;
35
36    public int getLength() {
37        return length;
38    }
```



```

39     }
40
41     public void setLength(int length) {
42         this.length = length;
43     }
44
45 }

```

2.2.4 依赖倒置原则 (DIP)

依赖倒置原则的含义是：高层模块不应该依赖于低层模块。两者都应该依赖于抽象。抽象不应该依赖于细节。细节应该依赖于抽象。

在模块编程中要依赖抽象编程，不要依赖于具体细节编程，即针对接口编程，不要针对具体实现编程。下面看一个使用集合类的例子，请仔细体会面向接口编程的含义。

代码片段7 DipTest.java

```

1  package cn.steven.oo.dip;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * 依赖倒置原则实例
8   */
9  public class DipTest {
10
11      public static void main(String[] args) {
12
13          /**
14           * 不符合依赖倒置原则的实例
15           */
16          ArrayList<String> lista = new ArrayList<String>();
17          lista.add("元素1");
18          lista.add("元素2");
19          lista.add("元素3");
20
21
22          /**
23           * 符合依赖倒置原则的实例
24           */
25          List<String> listb = new ArrayList<String>();
26          listb.add("元素1");
27          listb.add("元素2");
28          listb.add("元素3");
29      }
30
31 }

```

代码中不符合原则的代码带来的问题是如果以后将要使用另外一种List实现类，则lista这个变量就无法工作了，而listb变量还可以正常使用，所以编码时要依赖于稳定的抽象概念而不是

易变的具体类。

2.2.5 接口隔离原则 (ISP)

接口隔离原则的意图是：不要强迫客户依赖于它们不需要的方法，应用接口将两者隔离。

在具体的编程过程中一个类对另外一个类的依赖性应当是建立在最小的接口上的。如果客户端只需要某一些方法的话，那么就应当向客户端提供这些需要的方法，而不要提供不必要的方法。提供接口意味着向客户端做出承诺，过多的承诺会给系统的维护造成不必要的负担。提供接口的另一个意图是安全性，比如一个论坛系统中普通用户就无法获得一个用户的密码，而管理员就可以，这就对两者做了接口的隔离。

下面的例子说明了隔离的方法，假设有一个工厂，有两种角色分别是工人和消费者，工厂中既可以发生生产也可以发生消费活动，而工人只能生产，消费者只能消费，所以应将两者用接口隔离。

代码片段8 DipTest.java

```
1 package cn.steven.isp;
2
3 /**
4  * 符合接口隔离原则的工厂
5  */
6 public class Factory implements IProduce, IConsume {
7
8     @Override
9     public void produce() {
10         System.out.println("工人生产");
11     }
12
13     @Override
14     public void consume() {
15         System.out.println("消费者消费");
16     }
17
18     public static void main(String[] args) {
19
20         /**
21          * 创建消费者
22          */
23         IConsume consumer = new Factory();
24         consumer.consume();
25         //consumer.produce();    //无法执行
26         /**
27          * 创建生产者
28          */
29         IProduce worker = new Factory();
30         worker.produce();
31         //worker.consume();    //无法执行
32
33     }
```

```
34
35 }
36
37 /**
38  * 生产接口
39  */
40 interface IProduce {
41     public void produce();
42 }
43
44 /**
45  * 消费接口
46  */
47 interface IConsume {
48     public void consume();
49 }
```

由代码可见，通过双接口的设计，工人和消费者可以使用自己应该使用的方法而无法使用接口中未定义的方法，这样就提高了安全性和方便性。

2.3 Java面向对象的支持

Java通常被认为是一种完全面向对象的语言，但是由于其中有原始对象类型的存在，所以其实它并不纯粹。Java语言提供了面向对象编程的所有机制和特点：

- 抽象：面向对象程序设计的基本要素是抽象，程序员通过抽象来管理复杂性和可扩展性。
- 封装：封装是一种把代码和代码所操作的数据组装在一起，使这两者不受外界干扰和误用的机制。封装可被理解作为一种用做保护的包装器，以防止代码和数据被包装器外部所定义的其他代码任意访问。对包装器内部代码与数据的访问通过一个明确定义的接口来控制。封装代码的好处是每个人都知道怎样访问代码，进而无需考虑实现细节就能直接使用它，同时不用担心不可预料的副作用。

- 继承：Java只支持单继承，继承是代码复用最重要的方式之一。
- 多态：多态是指一个方法只能有一个名称，但可以有許多形态，也就是程序中可以定义多个同名的方法，可用“一个接口，多个方法”来描述。多态的实现分为重写和重载。

使用Java进行面向对象编程可以充分发挥面向对象的优点，由于其完全支持面向对象的特性，所以现在很多领域都使用Java作为主要的开发语言。

第3章 统一建模语言UML概述

统一建模语言（UML，Unified Modeling Language）是一个通用的可视化建模语言，用于对软件进行描述，可视化处理，构造和建立软件系统制品的文档。它是非专利的第三代建模和规约语言。

UML是一种开放的方法，用于说明、可视化、构建和编写一个正在开发的、面向对象的、软件密集系统的制品的开放方法。UML展现了一系列最佳工程实践，这些最佳实践在对大规模、复杂系统进行建模方面，特别是在软件架构层次方面已经被验证有效。

3.1 UML发展史

面向对象的分析与设计（OOA&D）方法的发展在20世纪80年代末至20世纪90年代出现了一个高潮，UML是这个高潮的产物。它不仅统一了Grady Booch、Jim Rumbaugh和Ivar Jacobson（这三个人是UML之父）的表示方法，而且对其做了进一步的发展，并最终统一为业界所接受的标准建模语言。

面向对象的建模语言出现于20世纪70年代中期，从1989年到1994年，其数量从不到十种增加到了五十多种，在众多的建模语言中，语言的创造者努力推崇自己的产品，并在实践中不断完善。但是，面向对象（OO）方法的用户并不了解不同建模语言的优缺点及其差异，因而很难根据自身应用特点选择合适的建模语言。20世纪90年代，一批新方法出现了，其中最引人注目的是Booch 1993、OOSE和OMT-2等。

Booch是面向对象方法最早的倡导者之一，他提出了面向对象软件工程的概念。1991年，他将以前面向Ada的工作扩展到整个面向对象设计领域。Booch 1993比较适合于系统的设计和构造。Rumbaugh等人提出了面向对象的建模技术（OMT）方法，采用了面向对象的概念，并引入各种独立于语言的表示符。这种方法用对象模型、动态模型、功能模型和用例模型，共同完成对整个系统的建模，所定义的概念和符号可用于软件开发的分析、设计和实现的全过程，软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2特别适用于分析和描述以数据为中心的信息系统。

Jacobson于1994年提出了OOSE方法，其最大特点是面向用例（Use-Case），并在用例的描述中引入了外部角色的概念。用例的概念是精确描述需求的重要武器，但用例贯穿于整个开发过程，包括对系统的测试和验证。OOSE比较适合用于商业工程和需求分析。此外，还有Coad/Yourdon方法，即著名的OOA&D，它是最早的面向对象的分析和设计方法之一。该方法简单、易学，适合于面向对象技术的初学者使用，但由于该方法在处理能力方面的局限，目前已很少使用。

概括来说，面对众多的建模语言，用户由于没有能力区别不同语言之间的差别，因此很难找到一种比较适合其应用特点的语言；其次，众多的建模语言实际上各有千秋；最后，虽然不同的建模语言大多类似，但仍存在某些细微的差别，这极大地妨碍了用户之间的交流。因此在客观上，极有必要在精心比较不同的建模语言优缺点及总结面向对象技术应用实践的基础上，

组织联合设计小组,根据应用需求,取其精华,去其糟粕,求同存异,统一建模语言。

1994年10月,Grady Booch和Jim Rumbaugh开始致力于这一工作。他们首先将Booch1993和OMT-2统一起来,并于1995年10月发布了第一个公开版本,称之为统一方法UM 0.8 (Unified Method)。同年,OOSE的创始人Ivar Jacobson加盟到这一工作。经过Booch、Rumbaugh和Jacobson三人的共同努力,1996年6月和10月他们分别发布了两个新的版本,即UML 0.9和UML 0.91,并将UM重新命名为UML (Unified Modeling Language)。1996年,一些机构将UML作为其商业策略的做法已日趋明显。UML的开发得到了来自公众的正面反应,并倡议成立了UML成员协会,以完善、加强和促进UML的定义工作。当时的成员有DEC、HP、I-Logix、Itellicorp、IBM、ICON Computing、MCI Systemhouse、Microsoft、Oracle、Rational Software、TI以及Unisys。这一机构对UML 1.0 (1997年1月)及UML 1.1 (1997年11月17日)的定义和发布起了重要的促进作用。

UML是一种定义良好、易于表达、功能强大且普遍适用的建模语言。它汇集了软件工程领域的新思想、新方法和新技术。它的作用域仅仅是支持面向对象的分析与设计,还支持从需求分析开始的软件开发的全过程。

在美国,至1996年10月,UML获得了工业界、科技界和应用界的广泛支持,已有700多个公司表示支持采用UML作为建模语言。1996年底,UML已稳占面向对象技术市场份额的85%,成为可视化建模语言事实上的工业标准。1997年11月17日,OMG采纳UML 1.1作为基于面向对象技术的标准建模语言。UML代表了面向对象方法的软件开发技术的发展方向,同时也提供软件工程化的思想和方法。

UML描述了一个系统的静态结构和动态行为。它将系统描述为一些离散的相互作用的对象并最终为外部用户提供一定功能的模型结构。静态结构定义了系统中的重要对象的属性和操作以及这些对象之间的相互关系。动态行为定义了对象的时间特性和对象为完成目标而相互进行通信的机制。从不同但相互联系的角度对系统建立的模型可用于不同的目的。

UML还包括可将模型分解成包的结构组件,用于软件小组将大的系统分解成易于处理的块结构,并理解和控制各个包之间的依赖关系,在复杂的开发环境中管理模型单元。它还包括用于显示系统实现和组织运行的组件。

UML不是一门程序设计语言,但可以使用代码生成器工具将UML模型转换为多种程序设计语言代码,或使用反向生成器工具将程序源代码转换为UML。

UML已经成为软件建模领域事实上的标准,它被成功地应用于许多领域,表达了众多不同的概念,例如:

- 程序语言实现 (Java、C++、Smalltalk、CORBA IDL等)的可视化表示;
- 直接可执行的模型 (如XUML);
- 与实现语言无关的软件规格说明;
- 高层架构和框架描述;
- 过程工程和重组;
- 网站结构;
- 工作流详述;
- 业务建模。

3.2 UML中的关系

UML定义的关系主要有六种：依赖、继承、关联、实现、聚合和组合。这些类间关系的理解和使用是掌握和应用UML的基础和关键。下面给出这六种主要UML关系的说明和类图描述。

3.2.1 依赖关系

依赖关系可以理解为一个类使用到了另一个类，而这种使用关系是具有偶然性的、临时性的、非常弱的。下面的例子中有A类和B类，A类依赖于B类，如图3-1所示。

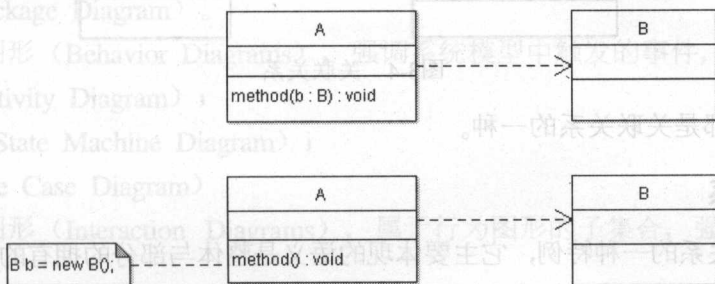


图3-1 依赖关系

依赖关系可以是方法参数的依赖，也可以是返回值的依赖，还可以是代码中的直接依赖。

3.2.2 继承关系

这种关系指的是一个类（称为子类）继承另外的一个类（称为父类）的功能，并可以增加它自己的新功能的能力。继承是面向对象设计中最常见的关系，它体现了一种is-a的关系；在Java中此类关系通过关键字**extends**明确标识，这是一种很强的耦合关系，如图3-2所示。

注意，类和类之间，接口和接口之间都可以继承；但是类只能单继承，接口可以多继承。

3.2.3 实现关系

它反映了一个类实现接口（可以是多个）的功能，实现是类与接口之间最常见的关系，在Java中此类关系通过关键字**implements**明确标识，如图3-3所示。

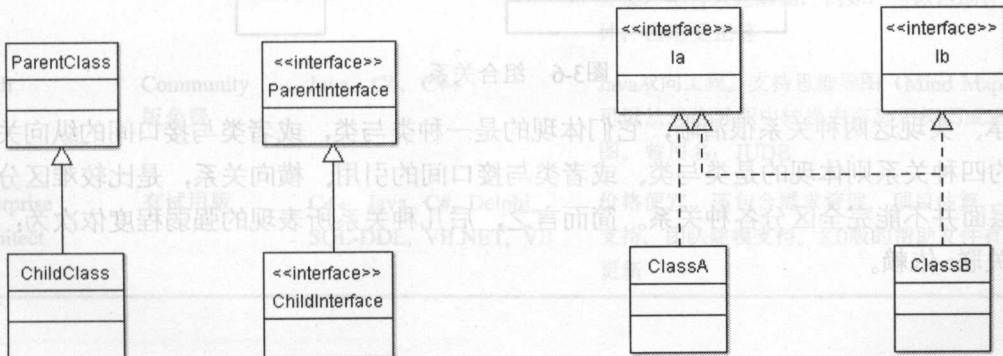


图3-2 继承关系

图3-3 实现关系

在Java中有很多接口是没有定义任何方法的，它们称作声明式接口。

3.2.4 关联关系

这种关系体现的是两个类、或者类与接口之间语义级别的一种强依赖关系，比如班级和学生。这种关系比依赖更强，不存在依赖关系的偶然性，关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。关联可以是单向、双向的；表现在代码层面为，被关联类B以类属性的形式出现在关联类A中，也可能是关联类A引用了一个类型为被关联类B的全局变量，如图3-4所示。

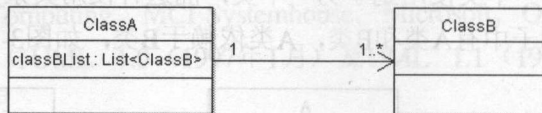


图3-4 关联关系

聚合和组合都是关联关系的一种。

3.2.5 聚合关系

聚合是关联关系的一种特例，它主要体现的语义是整体与部分的拥有的关系，即拥有一个（has-a）的关系，此时整体与部分之间是可分离的，它们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享，如图3-5所示。

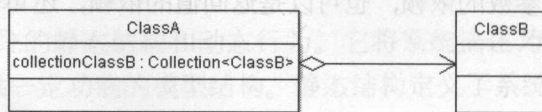


图3-5 聚合关系

3.2.6 组合关系

组合是关联关系的一种特例，它体现的是一种包含一个（contains-a）的关系，这种关系比聚合更强，也称为强聚合；它同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束，如图3-6所示。

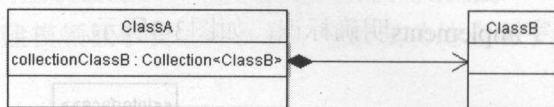


图3-6 组合关系

继承、实现这两种关系很清晰，它们体现的是一种类与类，或者类与接口间的纵向关系；而其他的四种关系则体现的是类与类、或者类与接口间的引用、横向关系，是比较难区分的，从代码层面并不能完全区分各种关系。简而言之，后几种关系所表现的强弱程度依次为：组合>聚合>关联>依赖。

3.3 UML中的图形类型

UML中定义的图形类型大体分为三类:

(1) 结构性图形 (Structure Diagrams), 强调的是系统式的建模, 包括:

- 类别图 (Class Diagram);
- 元件图 (Component Diagram);
- 复合结构图 (Composite structure Diagram);
- 部署图 (Deployment Diagram);
- 对象图 (Object Diagram);
- 套件图 (Package Diagram)。

(2) 行为式图形 (Behavior Diagrams), 强调系统模型中触发的事件, 包括:

- 活动图 (Activity Diagram);
- 状态机图 (State Machine Diagram);
- 用例图 (Use Case Diagram)。

(3) 沟通性图形 (Interaction Diagrams), 属于行为图形的子集合, 强调系统模型中的资料流程, 包括:

- 通信图 (Communication Diagram);
- 交互概述图 (Interaction Overview Diagram) (UML 2.0);
- 顺序图 (Sequence Diagram);
- 时间图 (UML Timing Diagram) (UML 2.0)。

3.4 UML工具软件

常用的UML工具软件有很多, 如表3-1所示。本书使用的工具是ArgoUML, 在具体工程使用时需要按照自己的要求来选择。

表3-1 常用UML工具

名称	许可证	支持语言	描述
ArgoUML	BSD	Java, C#	最早的开源UML工具, 支持OCL, 支持认知式开发, 不再只是画图, 例如, 可以自动评价设计、自动更正等
Astah	Community 版免费	Java, C#, C++	Java双向工程。支持思维导图 (Mind Map), 可以从思维导图中转换内容到用例图或者类图。曾用名: JUDE
Enterprise Architect	有试用版	C++, Java, C#, Delphi, SQL-DDL, VB.NET, VB	价格便宜, 还包含需求管理、项目估算、测试支持、团队建模支持。8.0版的帮助文件有大的更新

第4章 工厂方法模式 (Factory Method)

工厂方法模式定义了一个用户创建对象的接口，让子类决定实例化哪一个类。工厂方法模式使一个类的实例化延迟到其子类¹。

初看到工厂方法时可能有这样的疑问：

- 为什么要创建一个对象的接口？
- 为什么要把类实例化延迟到其子类？

先来看一个生活中的例子，比如某人有一个习惯，喜欢在睡前吃一个水果，无论什么水果都行。水果本身是一个抽象的概念，比如有苹果、桔子、西瓜、葡萄等。此人的习惯是任意水果都行，所以他不关心是哪一种水果的具体实现，如果让他执行每晚固定吃水果的程序时都确定是哪一种水果，这是不现实的，每天家里能有的水果是不可知的，这就是工厂方法模式要解决的问题。

在编程活动中，对于一个大型的软件工程，一个行之有效的方案就是进行模块式的分解。每一个模块由大量的代码构成，此模块的功能被其他模块使用，使用的方式就是：

- 实例化此模块（在面向对象的设计中）。
- 调用此模块功能。

注意这两个步骤中的第一步初看没什么问题，但是复杂的应用中就会产生严重的副作用。模块的接口虽然不易变，但是各种具体实现的类及其需求是经常变化的。比如编码客户端的时候设计的用户按钮是方形的，但是交付客户的时候客户的需求更改成了圆形，程序中创建的按钮有数万个，更改程序时就成了非常麻烦的事。

以上各种情形下所出现的棘手问题都可以由工厂方法模式解决。

4.1 商品上架遇到的问题

超市中有很多商品，每种商品上架时都需要合适的货架来摆放。由于商品的摆放非常烦琐，所以超市的领导决定使用软件实现商品上架。

在以下的设计中，基本思路是为商品选择货架，而不是为货架选择商品，这是根据超市的需求决定的设计方式。商品上架的流程是先拿到商品对象，创建一个可以容纳的货架空间，最后将商品放入。

初次看到这个需求时，大部分程序员会感觉非常简单，写出代码也很容易，示意代码如下：
商品类：

代码片段1 Goods.java

```
1 package cn.steven.pattern.demo.factorymethod.quest;  
2 /**
```

¹Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.-GOF95

(续表)

```

3  * 商品
4  */
5  public class Goods {
6  /**
7   * 商品的名字
8   */
9   private String name;
10
11  public String getName() {
12      return name;
13  }
14
15  public void setName(String name) {
16      this.name = name;
17  }
18
19  public Goods(String name) {
20      this.name = name;
21  }
22  }

```

• 货架类:

代码片段2 SampleShelf.java

```

1  package cn.steven.pattern.demo.factorymethod.quest;
2
3  /**
4   * 货架
5   */
6  public class SampleShelf {
7
8      private String name = "普通货架";
9
10     public void put(Goods goods) {
11         /**
12          * 打印货架名, 货架类信息, 所放商品信息
13          */
14         System.out.println(this.name + "(" + this + ")" + "上摆放了"
15             + goods.getName());
16     }
17
18  }

```

• 运行类:

代码片段3 SampleShelfClient.java

```

1  package cn.steven.pattern.demo.factorymethod.quest;
2
3  /**
4   * 测试类
5   */

```

```
6 public class SampleShelfClient {
7
8     public static void main(String[] args) {
9         // 得到商品
10        Goods goods1 = new Goods("矿泉水");
11        // 得到货架
12        SampleShelf shelf1 = new SampleShelf();
13        // 上架
14        shelf1.put(goods1);
15
16        // 得到商品
17        Goods goods2 = new Goods("方便面");
18        // 得到货架
19        SampleShelf shelf2 = new SampleShelf();
20        // 上架
21        shelf2.put(goods2);
22    }
23 }
```

在此例中可见是由代码片段3调用代码片段1和代码片段2来完成所有的功能的。

运行结果如图4-1所示。

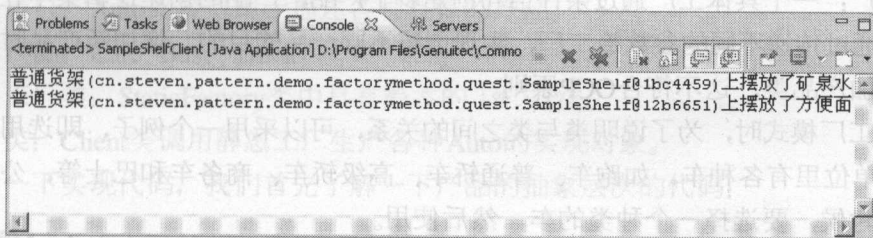


图4-1 SampleShelfClient.java运行结果

由以上代码和结果可以看出，设计和代码都非常直接明了，需要商品就实例化一个商品，需要货架就实例化货架，如代码片段3中10行~21行所示。由日志可见，两个商品分别取得了不同的货架进行了存放（两个实例的hashcode不同，一个为1bc4459，另一个为12b6651；不同的对象hashcode是不同的）。

但是我们不妨设想一下在这个系统设计出来之后发生的事。

- 货架的使用应该具有可复用性，即如果一个货架没有装满的话，还可以用来装其他的货物。由于使用程序来装货物，所以货物不应该知道货架的细节。在此例子中并没有实现此功能。

- 每一种货物所摆放的货架是有要求的，不能任意用一种货架。

- 当使用代码大量增加时，如果再想做统一的操作，将会增加大量代码（考虑一下上例中若想对货架存放货物前都进行清理动作的情况）。

- 未来将会增加货架的种类，在此系统中，增加货架的种类势必增加货架类和改变使用的代码，使用代码和特定的货架是紧密耦合的。

由以上研究可以发现，简单的设计在需求的复杂性提高了后会出现问题，在本章中所学习的工厂方法模式就是解决创建类问题的一种方法。

4.2 工厂方法模式的结构

工厂方法模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。这样，客户端代码就不会直接耦合需要的类，而是通过一个工厂类来耦合，从而改善了模块之间的依赖。

常见的工厂模式有如下三种：

- 简单工厂模式/静态工厂方法模式。
- 工厂方法模式/多态性工厂模式/虚拟构造子模式。
- 抽象工厂模式/工具箱模式。

在本章中，主要讲解前两种模式，下一章着重讲解第三种工厂模式。

4.2.1 简单工厂模式

简单工厂模式是最易理解的工厂模式，它的一个别名是静态工厂方法模式。当需要一个相同的接口，但其实现具有不同的功能的时候，就可以用一个工厂模式来产生其中多个类的一个实例。

简单工厂：一个具体工厂通过条件语句创建多个产品，产品的创建逻辑集中在一个工厂类上。客户端通过传递不同的参数给工厂，实现创建不同产品的目的。增加新产品时，需要修改工厂类、增加产品类，这不符合OCP原则。

在讲解工厂模式时，为了说明类与类之间的关系，可以采用一个例子，即选用一个单位用车的例子，单位里有各种车，如跑车、普通轿车、高级轿车、商务车和巴士等，公司中的员工需要用车的时候，要选择一种种类的车，然后使用。

先来看一下通常所见到的代码初步设计类图，如图4-2所示。

图4-2使用的结构非常类似于代码片段3，这种设计所能造成的缺点我们也已经讨论过了，下面来看一下由此图进行的模式演化，如图4-3所示。

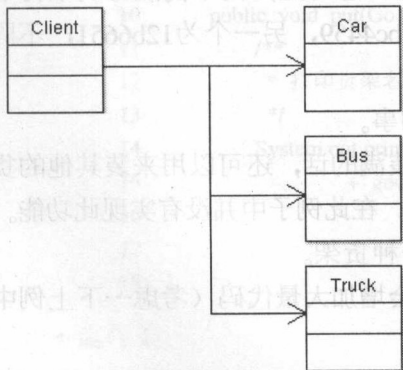


图4-2 初步设计类图

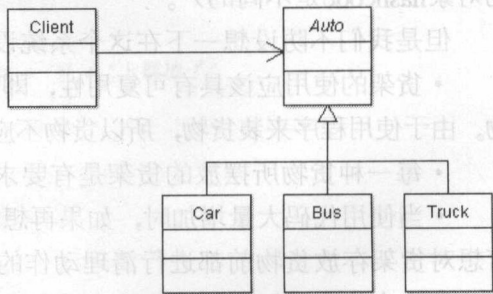


图4-3 抽象演化

由图4-3可见，设计中加入了一个抽象的层次Auto类，加入此抽象层次的好处是显而易见的，它帮助架构降低了Client类和各种具体车型之间的耦合度，而且符合了开一闭原则，如果以后需要加入其他车型，设计将变得很容易。

但是，此设计的缺点也是存在的，它并没有将客户端的构建具体依赖对象的代码和客户端

的使用代码完全分离开，这样就会对扩展性设计和功能性设计造成麻烦，请思考以下问题：

- 客户所用的任何一部车都需要在使用前登记。
- 如果Car类车型没有了，使用Truck车型替代。
- 公司的Car类车全部换为SuperCar。

如果采用简单工厂模式，则可以解决以上问题，其类图如图4-4所示。

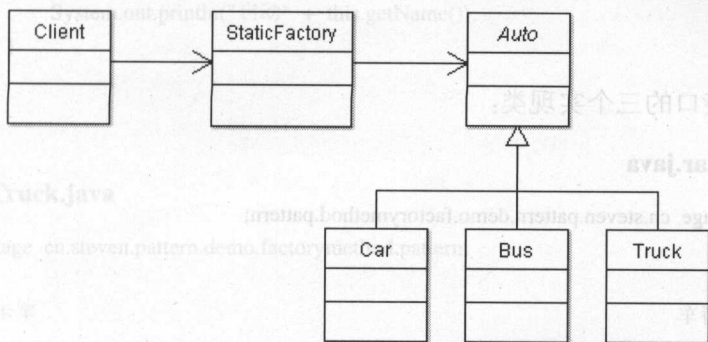


图4-4 简单工厂模式类图

图4-4中的各种元素如下：

- 产品接口：Auto此接口用于描述各种具体的产品。
- 具体产品：Car、Bus、Truck。
- 静态工厂类：StaticFactory类中具有静态的工厂方法。
- 客户类：Client类调用静态工厂生产各种Auto的实现对象。

下面看一下实现代码，我们首先了解一下产品的抽象层次的代码：

代码片段4 Auto.java

```
1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 抽象汽车类
5  */
6 public abstract class Auto {
7     /**
8      * 车型名称
9      */
10    private String name;
11
12    /**
13     * 抽象的工作方法
14     */
15    abstract public void run();
16
17    /**
18     * 获得车型名称方法
19     */
20    public String getName() {
21        return name;
22    }
```

```

23
24 /**
25  * 设置车型名称方法
26  */
27 public void setName(String name) {
28     this.name = name;
29 }
30
31 }

```

下面是Auto接口的三个实现类:

代码片段5 Car.java

```

1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 轿车
5  */
6 public class Car extends Auto {
7
8     /**
9      * 构造方法
10     */
11     public Car() {
12         this.setName("轿车");
13     }
14
15     /**
16      * 重写run方法
17     */
18     @Override
19     public void run() {
20         System.out.println("启动"+this.getName());
21     }
22
23 }

```

代码片段6 Bus.java

```

1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 巴士
5  */
6 public class Bus extends Auto {
7
8     /**
9      * 构造方法
10     */
11     public Bus() {
12         this.setName("巴士");

```



```
13     }
14
15     /**
16      * 重写run方法
17      */
18     @Override
19     public void run() {
20         System.out.println("启动" + this.getName());
21     }
22
23 }
```

代码片段7 Truck.java

```
1 package cn.steven.pattern.demo.factorymethod.pattern;
2 /**
3  * 卡车
4  */
5 public class Truck extends Auto{
6
7     /**
8      * 构造方法
9      */
10    public Truck() {
11        this.setName("卡车");
12    }
13
14    /**
15     * 重写run方法
16     */
17    @Override
18    public void run() {
19        System.out.println("启动" + this.getName());
20    }
21
22 }
```

下面看一下简单工厂模式的重点类:

代码片段8 StaticFactory.java

```
1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 静态工厂
5  */
6 public class StaticFactory {
7
8     /**
9      * 工厂方法
10     * @param autoId 车型编号
11     * @return 具体车对象
```



```

12     */
13     public static Auto createAuto(int autoId) {
14         switch (autoId) {
15             case 1:
16                 return new Car();
17             case 2:
18                 return new Bus();
19             case 3:
20                 return new Truck();
21             default:
22                 /**
23                  * 如果没有此种车型，抛出运行时异常
24                  */
25                 throw new RuntimeException("没有这种车型!");
26             }
27         }
28     }

```

由代码片段8可见，简单工厂模式就是使用了一个简单的静态方法来创建对象，创建时可以依据参数决定要创建的对象所属的类，也可以通过固定实现类的方法创建对象。还有比较灵活的方法，比如使用外部配置文件等的方式来实现创建对象。

代码片段9 Client.java

```

1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 客户端类
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9         /**
10          * 调用静态工厂创建对象
11          */
12         Auto auto1 = StaticFactory.createAuto(1);
13         auto1.run();
14
15         Auto auto2 = StaticFactory.createAuto(2);
16         auto2.run();
17
18         Auto auto3 = StaticFactory.createAuto(4);
19         auto3.run();
20     }
21
22 }

```

以上Client.java的执行结果如图4-5所示。

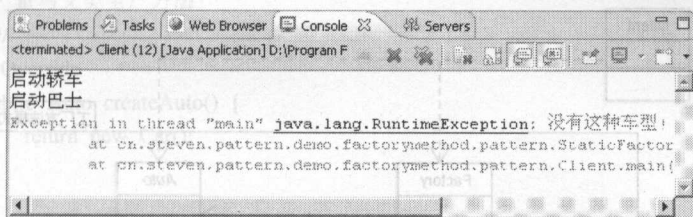


图4-5 Client.java执行结果

注意图4-5中的异常是代码片段8中第25行触发的。

经过了以上的简单工厂模式的设计，可以发现它有以下几个方面的优点：

- 用代码和具体使用类的耦合度降低。
- 令创建与使用的代码相分离，可以独立地变化，易维护和扩展。
- 可以通过外部配置的方法将耦合度进一步降低。

但是简单工厂模式也有一定的缺点：

- 当产品类有复杂的多层次等级结构时，工厂类却只有一个，只能以不变应万变，就是模式的缺点。

- 这个工厂类集中了所有的产品创建逻辑，形成一个无所不知的全能类，有人把这种类叫做上帝类（God Class）。如果这个全能类不能正常工作了，整个程序都会受到影响。

- 将这么多的逻辑集中放在一个类里面当产品类有不同的接口种类时，工厂类需要判断在什么时候创建某种产品。这种对时机的逻辑判断和对哪一种具体产品的逻辑判断混合在一起，使得系统在将来进行功能扩展时较为困难。

- 由于简单工厂模式使用静态方法作为工厂方法，而静态方法无法由子类继承，因此工厂角色无法形成基于继承的等级结构，工厂的可扩展性受到限制。

上述缺点在工厂方法模式中 will 得到改进。

4.2.2 工厂方法模式

通过上一节的学习，我们已经掌握了简单工厂模式的实现方法，但是发现了它的几点不足之处，其缺点可以总结为：

- 工厂实现的产品抽象，不具备复杂性。
- 工厂不具备抽象性。

如何解决这些问题呢？GoF的设计模式中提出了工厂方法模式以解决它的缺点。

解决的方法就是将各种产品使用不同的工厂来生产，各种工厂中的创建方法可以互相独立地改变，并将这些工厂类抽象出一个共同的父类，具体类图如图4-6所示。

如图4-6所示，工厂方法和简单工厂的区别就在于其对工厂也做了一层抽象，不同的工厂负责不同的产品的生产。

在此例子中，Auto汽车抽象的代码并没有改变，相关代码参见：

- 代码片段4 Auto.java
- 代码片段5 Car.java
- 代码片段6 Bus.java
- 代码片段7 Truck.java

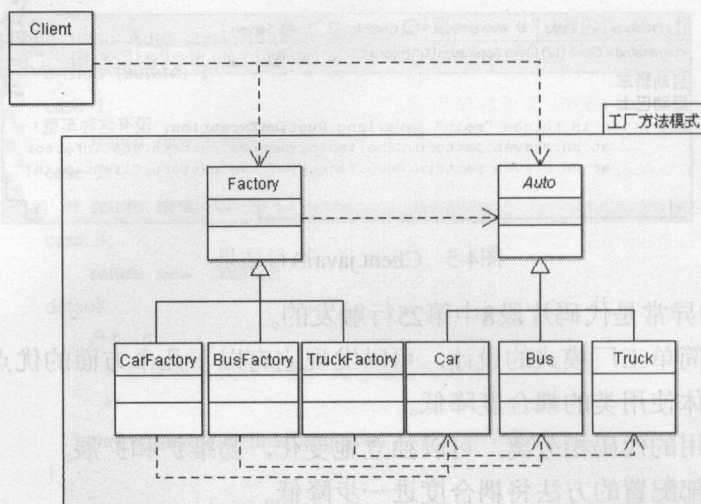


图4-6 工厂方法模式类图

下面展示的是工厂的抽象层次代码:

代码片段10 Factory.java

```

1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 工厂方法模式
5  */
6 public abstract class Factory {
7
8     /**
9      * 抽象生产方法，将被子类重写
10     * @return Auto类的子类实例
11     */
12     abstract public Auto createAuto();
13 }
  
```

注意代码片段10是一个抽象类，它的功能是给具体的工厂实现类提供一个抽象，这样客户代码就可以依赖此抽象而不用依赖具体的工厂类。注意其代码第12行的返回值需要是一个产品的抽象层次的描述，而不能是具体的某种产品。

具体实现类如下面代码所示。

轿车工厂类:

代码片段11 CarFactory.java

```

1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 具体工厂类
5  */
6 public class CarFactory extends Factory {
7
8     /**
  
```



```
9      * 重写父类生产方法
10     */
11     @Override
12     public Auto createAuto() {
13         return new Car();
14     }
15
16 }
```

巴士工厂类:

代码片段12 BusFactory.java

```
1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 具体工厂类
5  */
6 public class BusFactory extends Factory {
7
8     /**
9      * 重写父类生产方法
10     */
11     @Override
12     public Auto createAuto() {
13         return new Bus();
14     }
15
16 }
```

卡车工厂类:

代码片段13 TruckFactory.java

```
1 package cn.steven.pattern.demo.factorymethod.pattern;
2
3 /**
4  * 具体工厂类
5  */
6 public class TruckFactory extends Factory {
7
8     /**
9      * 重写父类生产方法
10     */
11     @Override
12     public Auto createAuto() {
13         return new Truck();
14     }
15
16 }
```

客户端代码:

代码片段14 Client2.java

```

1  package cn.steven.pattern.demo.factorymethod.pattern;
2
3  /**
4   * 客户端类
5   */
6  public class Client2 {
7
8      public static void main(String[] args) {
9          /**
10           * 调用工厂方法创建不同对象
11           */
12
13          /**
14           * 创建变量（使用抽象类描述，面向接口编程）
15           */
16          Factory factory;
17          Auto auto;
18
19          /**
20           * 创建Car
21           */
22          factory = new CarFactory();
23          auto = factory.createAuto();
24          auto.run();
25
26          /**
27           * 创建Bus
28           */
29          factory = new BusFactory();
30          auto = factory.createAuto();
31          auto.run();
32
33          /**
34           * 创建Truck
35           */
36          factory = new TruckFactory();
37          auto = factory.createAuto();
38          auto.run();
39
40      }
41
42  }

```

运行结果如图4-7所示。

以上就是工厂方法模式的简单实现，分析其实现可以发现简单工厂模式的两种主要缺点已经被解决了，为了更容易理解其优点，下面举例说明工厂方法能实现而简单工厂不能实现或不容易实现的情景：

- 对每一种生产出的产品实例做不同的操作。

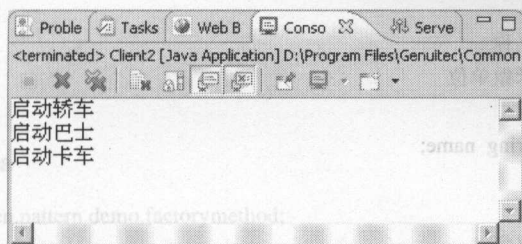


图4-7 Client2.java运行结果

- 增加一种完全不同类型的产品。
- 在不修改原有工厂类的情况下增加对新产品的支持。

4.2.3 使用工厂方法模式解决商品上架问题

经过了两种工厂模式的学习，现在可以思考一下商品上架的设计问题了，由代码片段1、代码片段2和代码片段3可见，最初的设计方案无法满足自动化商品上架的需求，其原因在于：

- 上架时需要由程序员指定创建货架的对象，缺乏灵活性。
- 如果需要对所有货架在使用时加以处理是不方便的。
- 如果货架的种类增加，需要修改大量的客户端代码。
- 如果需要增加商品的种类不方便，而且客户端代码需要大量修改。

如下设计中将采用工厂方法模式进行设计，用以改进原始设计的缺点，如图4-8所示。

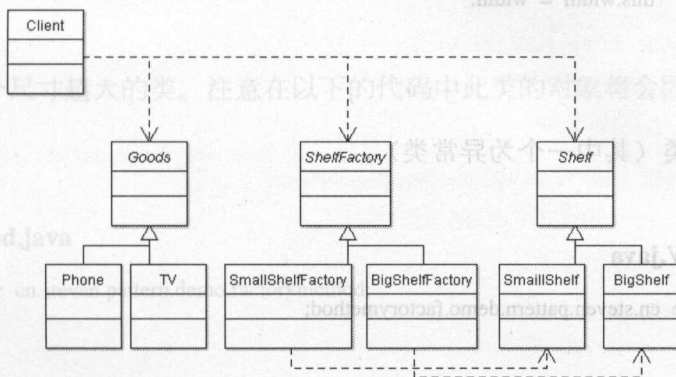


图4-8 使用工厂方法模式解决商品上架问题

图4-8中Goods、ShelfFactory和Shelf是一些平行的层次类，我们要通过代码使之有效组织起来。

首先来看一下货物的代码实现：

代码片段15 Goods.java

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 货物抽象类
5  */
6 public abstract class Goods {
7
```



```

8      /**
9       * 货物名称
10      * 以厘米做单位
11      */
12     private String name;
13
14     /**
15     * 货物宽度
16     */
17     private int width;
18
19     public String getName() {
20         return name;
21     }
22
23     public void setName(String name) {
24         this.name = name;
25     }
26
27     public int getWidth() {
28         return width;
29     }
30
31     public void setWidth(int width) {
32         this.width = width;
33     }
34
35 }

```

此类有三个子类（其中一个为异常类）。

电视机类：

代码片段16 TV.java

```

1  package cn.steven.pattern.demo.factorymethod;
2
3  /**
4   * 一个具体的商品
5   */
6  public class TV extends Goods {
7
8      /**
9       * 构造方法
10     */
11     public TV() {
12         //调用父类默认构造方法
13         super();
14         //设置名称
15         this.setName("电视机");
16         //设置宽度
17         this.setWidth(100);
18     }

```

19 20 SmallShelf.java

20 }

电话类:

代码片段17 Phone.java

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 一个具体的商品
5  */
6 public class Phone extends Goods {
7
8     /**
9      * 构造方法
10     */
11     public Phone() {
12         //调用父类默认构造方法
13         super();
14         //设置名称
15         this.setName("电话机");
16         //设置宽度
17         this.setWidth(20);
18     }
19
20 }
```

这里多设计一个尺寸超大的类。注意在以下的代码中此类的对象将会因没有合适的货架而发生异常。

双人床类:

代码片段18 Bed.java

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 一个具体的商品
5  */
6 public class Bed extends Goods {
7
8     /**
9      * 构造方法
10     */
11     public Bed() {
12         //调用父类默认构造方法
13         super();
14         //设置名称
15         this.setName("双人床");
16         //设置宽度
17         this.setWidth(300);
18     }
19 }
```

19

20 }

下面展示的是货架层次的各个类。

货架抽象类:

代码片段19 Shelf.java

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 货架抽象类
5  */
6 public abstract class Shelf {
7
8     /**
9      * 货架容纳货物的最大宽度
10     * 以cm做单位
11     */
12     private int maxWidth;
13
14     /**
15      * 货架名称
16      */
17     private String shelfName;
18
19     public String getShelfName() {
20         return shelfName;
21     }
22
23     public void setShelfName(String shelfName) {
24         this.shelfName = shelfName;
25     }
26
27     public int getMaxWidth() {
28         return maxWidth;
29     }
30
31     public void setMaxWidth(int maxWidth) {
32         this.maxWidth = maxWidth;
33     }
34
35     public void put(Goods goods) {
36         System.out.println("将 " + goods.getName() + " 放入 "
37             + this.getShelfName());
38     }
39 }
```

此类有两个子类。

小型货架 (90cm) :

代码片段20 SmallShelf.java

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 一种具体的货架
5  */
6 public class SmallShelf extends Shelf {
7
8     /**
9      * 构造方法
10     */
11     public SmallShelf() {
12         // 调用父类构造方法
13         super();
14         this.setShelfName("一种90cm的小货架");
15         this.setMaxWidth(90);
16     }
17
18 }
```

大型货架 (200cm) :

代码片段21 BigShelf.java

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 一种具体的货架
5  */
6 public class BigShelf extends Shelf {
7
8     /**
9      * 构造方法
10     */
11     public BigShelf() {
12         // 调用父类构造方法
13         super();
14         this.setShelfName("一种200cm的大货架");
15         this.setMaxWidth(200);
16     }
17
18 }
```

接下来看一下工厂的实现。

货架抽象工厂:

代码片段22 ShelfFactory.java

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 货架抽象类
```

```

5  */
6  public abstract class ShelfFactory {
7
8      /**
9       * 所生产的货架容纳货物的最大宽度
10      * 以厘米做单位
11      */
12      private int maxWidth;
13
14      public int getMaxWidth() {
15          return maxWidth;
16      }
17
18      public void setMaxWidth(int maxWidth) {
19          this.maxWidth = maxWidth;
20      }
21
22      /**
23       * 抽象生产方法
24       * @return 具体的货架对象
25       */
26      abstract public Shelf createShelf();
27
28  }

```

下面的两个实现类与生产的两种货架一一对应。

小型货架工厂：

代码片段23 SmallShelfFactory.java

```

1  package cn.steven.pattern.demo.factorymethod;
2
3  /**
4   * 一种具体的工厂类
5   */
6  public class SmallShelfFactory extends ShelfFactory {
7
8      public SmallShelfFactory() {
9          super();
10         //设置最大宽度
11         this.setMaxWidth(new SmallShelf().getMaxWidth());
12     }
13
14     /**
15      * 重写生产方法
16      */
17     @Override
18     public Shelf createShelf() {
19         Shelf s = new SmallShelf();
20         return s;
21     }

```

大型货架工厂:

代码片段24 BigShelfFactory.java

```

1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 一种具体的工厂类
5  */
6 public class BigShelfFactory extends ShelfFactory {
7
8     public BigShelfFactory() {
9         super();
10        // 设置最大宽度
11        this.setMaxWidth(new BigShelf().getMaxWidth());
12    }
13
14    /**
15     * 重写生产方法
16     */
17    @Override
18    public Shelf createShelf() {
19        Shelf s = new BigShelf();
20        return s;
21    }
22
23 }
```

客户在使用Goods、Shelf和ShelfFactory这三个抽象层次时,会发现使用起来比较复杂,因为需要人为去选择要使用哪种具体的工厂,下面展示的是一个助手类,此类的功能是连接Goods和ShelfFactory这两个层次的关系。

助手类:

代码片段25 PlaceGoodsHelper.java

```

1 package cn.steven.pattern.demo.factorymethod;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 放置货物的助手类,此类是Goods和Shelf的桥梁类
8  */
9 public class PlaceGoodsHelper {
10
11     /**
12      * 放置方法,此方法需要进行同步处理
13      *
14      * @param goods 所需放置的货物
```



```

15  */
16  public synchronized static void placeGoods(Goods goods) {
17      /**
18       * 将所有的货架工厂放置在列表中
19       * 按照尺寸从小到大依次放置
20       * 注意下面的代码使用了泛型
21       */
22      List<ShelfFactory> shelfList = new ArrayList<ShelfFactory>();
23      shelfList.add(new SmallShelfFactory());
24      shelfList.add(new BigShelfFactory());
25
26      /**
27       * 货架变量
28       */
29      Shelf rightShelf;
30
31      /**
32       * 动态判断需要哪种货架
33       * 增强for循环
34       */
35      for (ShelfFactory sf : shelfList) {
36          /**
37           * 货架合适
38           */
39          if (sf.getMaxWidth() >= goods.getWidth()) {
40              /**
41               * 实例化特定的货架
42               * 放置货物
43               */
44              try {
45                  /**
46                   * 使用工厂创建实例
47                   */
48                  rightShelf = sf.createShelf();
49                  rightShelf.put(goods);
50                  //如果有合适的货架，则在放置后退出方法
51                  return;
52              } catch (Exception e) {
53                  e.printStackTrace();
54              }
55          }
56      }
57
58      //运行到这里说明没有合适的货架
59      throw new RuntimeException("没有找到符合"+goods.getName()+
60      "尺寸的货架!");
61
62  }
63
64  }

```

注意代码片段25中使用了JDK5¹及其以上版本的新功能：泛型²、增强for循环³。

最后看一下客户端的代码：

代码片段26 Client.java

```

1 package cn.steven.pattern.demo.factorymethod;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 客户端类
8  */
9 public class Client {
10
11     public static void main(String[] args) {
12
13         /**
14          * 初始化一些商品放置于列表中
15          */
16         List<Goods> goodsList = new ArrayList<Goods>();
17
18         goodsList.add(new TV());
19         goodsList.add(new Phone());
20         goodsList.add(new Bed());
21
22         System.out.println("初始化货物列表完成！");
23
24         /**
25          * 放置商品
26          */
27         for (Goods goods : goodsList) {
28             try {
29                 PlaceGoodsHelper.placeGoods(goods);
30             } catch (Exception e) {
31                 e.printStackTrace();
32             }
33         }
34
35         System.out.println("放置货物完成！");
36     }
37
38 }

```

由代码片段26可见，现在客户放置商品的操作变得很简单，只需要调用助手类放置商品即可，不需要考虑商品的大小和具体货架的类型。

¹http://java.sun.com/javase/downloads/index_jdk5.jsp.

²<http://baike.baidu.com/view/965887.htm>.

³http://www.java2s.com/Tutorial/Java/0140_Collections/forstatementforIterableobjectinJDK5hasbeenenhanced.htm.

Client.java的运行结果如图4-9所示。



图4-9 Client.java运行结果

到此，我们已经可以使用工厂方法模式完全解决商品上架的问题了，但是，由于篇幅原因，有一些因素没有完全考虑到，读者可以自行扩展。部分扩展如下：

- 考虑一个货架可以放多个货物的情况。
- 考虑商品放置的优先级的问题。
- 考虑更多类型的货架，如不同的形状和材质的货架。
- 考虑货物体积与货架空间相互作用引发的最优化放置问题。

4.2.4 工厂模式在JDK中的实例

工厂方法模式在Java程序设计中非常常见，经常可以看到在Java类中有factory的字样，但是有factory字样的类有的是工厂方法模式，有的是抽象工厂模式，要注意区别。

注意下例中是两个工厂方法的具体应用：

代码片段27 FactoryDemo.java

```
1 package cn.steven.pattern.demo.factorymethod.jdk;
2
3 import javax.xml.parsers.DocumentBuilder;
4 import javax.xml.parsers.DocumentBuilderFactory;
5
6 /**
7  * JDK中工厂方法的使用
8  */
9 public class FactoryDemo {
10
11     public static void main(String[] args) throws Exception {
12
13         /**
14          * 工厂方法模式创建DocumentBuilder
15          */
16         DocumentBuilderFactory builderFactory =
17             DocumentBuilderFactory.newInstance();
18         DocumentBuilder documentBuilder =
19             builderFactory.newDocumentBuilder();
20
21         /**
22          * 工厂方法模式创建布尔类型的实例
23          */
```



```
24 Boolean v = Boolean.valueOf(true);
25
26 }
27
28 }
```

注意上例中第24行看似不是工厂方法，但是研究过JDK的代码后就会发现是工厂方法。

代码片段28 Boolean.java

```
1 package java.lang;
2 public final class Boolean implements java.io.Serializable,
3                                     Comparable<Boolean>
4 {
5     public static final Boolean TRUE = new Boolean(true);
6     public static final Boolean FALSE = new Boolean(false);
7
8     public Boolean(boolean value) {
9         this.value = value;
10    }
11
12    public Boolean(String s) {
13        this(toBoolean(s));
14    }
15
16    public static boolean parseBoolean(String s) {
17        return toBoolean(s);
18    }
19
20    public boolean booleanValue() {
21        return value;
22    }
23
24    public static Boolean valueOf(boolean b) {
25        return (b ? TRUE : FALSE);
26    }
27
28    public static Boolean valueOf(String s) {
29        return toBoolean(s) ? TRUE : FALSE;
30    }
31
32    public static String toString(boolean b) {
33        return b ? "true" : "false";
34    }
35
36    public String toString() {
37        return value ? "true" : "false";
38    }
39    ...
40 }
```

注意代码片段28中第5行~第6行定义了两个静态的对象，当使用代码第24行~第26行时就

会返回这两者之一。因为Boolean类型只有两种类型的值，所以这样的设计是很好的。

4.2.5 工厂方法模式的使用范围

工厂方法模式使用的范围如下：

- 当客户程序不需要知道要使用对象的创建过程时。
- 客户程序使用的对象存在变动的可能，或者根本就不知道使用哪一个具体的对象时。
- 当客户想分离对象的创建和使用代码时。
- 当客户想集中管理创建代码时。
- 当客户需要使用的对象种类非常多，并且有扩展需求时。

其优缺点分析如下：

优点

工厂方法模式使用继承自抽象工厂角色的多个子类来代替简单工厂模式中的“上帝类”。正如上面所说，这样便分担了对象承受的压力；而且这样使结构变得灵活起来——当有新的产品产生时，只要按照抽象产品角色、抽象工厂角色提供的合同来生成，那么代码就可以被客户使用了，而不用去修改任何已有代码。可以看出工厂模式也是符合开一闭原则的。

缺点

可以看出工厂方法的加入，使得对象的数量成倍增长。当产品种类非常多时，会出现大量的与之对应的工厂对象，这不是我们所希望的。如果不能避免这种情况，可以考虑使用简单工厂模式与工厂方法模式相结合的方式减少工厂类，即对于产品树上类似的种类（一般是树的叶子节点元素互为兄弟的情况）使用简单工厂模式来实现。

4.2.6 简单工厂与其他模式的区别

工厂方法模式与简单工厂模式的区别如下：

- 工厂方法模式与简单工厂模式在结构上的不同不是很明显。工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。
- 工厂方法模式之所以有一个别名叫多态性工厂模式是因为，具体工厂类都有共同的接口，或者有共同的抽象父类。
- 当系统扩展需要添加新的产品对象时，仅仅需要添加一个具体对象以及一个具体工厂对象，原有工厂对象不需要进行任何修改，也不需要修改客户端，这很好地符合了“开放—封闭”原则。而简单工厂模式在添加新产品对象后不得不修改工厂方法，扩展性不好。
- 工厂方法模式退化后可以演变成简单工厂模式。

4.3 工厂方法模式总结

工厂方法模式是设计模式中应用最为广泛的模式之一，通过本文，相信读者已经对它有了一定的认识。然而我们要明确的是：在面向对象的编程中，对象的创建工作非常简单，但对象的创建时机却很重要。工厂方法模式要解决的就是对象的创建时机问题，它提供了一种扩展的策略，这很好地符合了开放—封闭原则。

第5章 抽象工厂模式 (Abstract Factory)

抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无需为它们指定具体的类。¹

抽象工厂模式是一种在动态语言中不太常用的创建型设计模式，它的别名是kit，在软件系统中，经常面临着“一系列相互依赖的对象”的创建工作。同时，由于需求的变化，往往存在着更多系列对象的创建工作。那么，如何应对这种变化？如何绕过常规的对象创建方法（new），提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？这就是本章要讲的抽象工厂模式所能解决的问题。

抽象工厂模式也属于创建模式，也是关于如何创建对象的模式，它是对工厂方法模式的拓展和延伸。工厂方法模式主要是针对任意数量的产品等级的，多应用于虚拟构造子类，抽象工厂则可以处理多个产品族结构。

要理解抽象工厂模式，最关键的就是要理解“产品族”的概念。日常生活中也有很多此概念的例子，比如，中国古代的人和现代的人都要写字，都需要“笔墨纸砚”，古人用的纸是宣纸，现代人用信纸；古人用毛笔，现代人用钢笔；古人用墨汁，现代人用墨水。虽然毛笔和钢笔的功能一样，但是却是在两个完全不同的场合下使用，这里，这两种笔就是一个“产品族”。

在软件环境中，经常可以看到软件以不同的视觉感受展现的例子，以下以JDK自带的SwingSet2²为例展示视觉感受带来的不同，如图5-1、图5-3和图5-3所示。

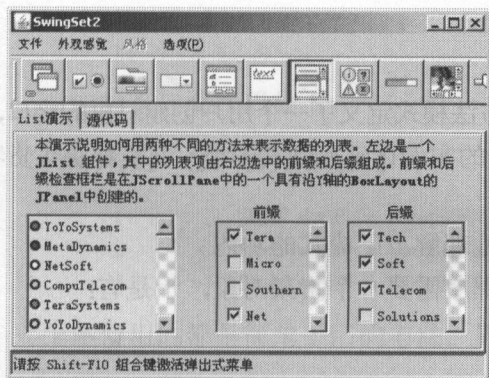


图5-1 SwingSet2 Windows 视觉
(Look&Feel)

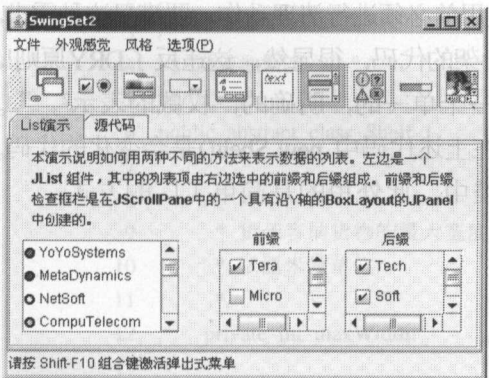


图5-2 SwingSet2 Java 视觉
(Look&Feel)

¹Provide an interface for creating families of related or dependent objects without specifying their concrete classes. GOF[95]

²%JAVA_HOME%\demo\jfc\SwingSet2.

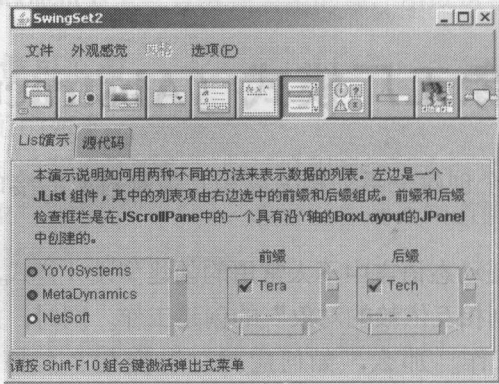


图5-3 SwingSet2 Motif视感 (Look&Feel)

由上述三幅图片可以看出，功能一样的组件能带给我们不同的感受。这就是非常典型的产品族问题，如上述图片中的一些按钮的显示是不同的。

5.1 深入探讨商品上架遇到的问题

在第4章中我们已经学习了工厂方法模式，工厂方法模式定义了一个用户创建对象的接口，让子类决定实例化哪一个类。工厂方法模式使一个类的实例化延迟到其子类，并且解决了货物上架的问题。

下面我们梳理一下模式的演化过程，以便更好地理解工厂模式的功能。

最初的需求是将货物放入货架，所以，客户端程序需要一个货架对象，于是有：

代码片段1 工厂模式演化1

```
1 Shelf shelf = new Shelf();
```

注意在上面的代码片段1中Shelf是一个具体的货架类。可以想象现在的客户代码中将会有很多这样的创建代码，但是如果有一种需求是在货架使用前必须进行清理动作，要满足这种需求，那么所有的这种创建代码后面就必须再加上一条清理货架的代码，很显然，这违反了DRY原则¹，此项原则的大意就是“每一个知识都必须具备系统内一个单一的，明确的，权威的表示²”。

面向对象设计还有一个原则就是“封装变化”，在上述代码中 new Shelf()是会变化的代码。于是可以把创建货架的代码独立放在另外一个类中，此处用的是简单工厂的方式：

代码片段2 工厂模式演化2_1

```
1 class SampleFactory{
2     public static synchronized Shelf createShelf(){
3         return new Shelf();
4     }
5 }
```

客户端代码演化为：

¹http://en.wikipedia.org/wiki/Don%27t_repeat_yourself.
²Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

代码片段3 工厂模式演化2_2

```
1 Shelf shelf2 = SampleFactory.createShelf();
```

如果没有增加各种货架的需求,那么使用上述解决方案就可以了,但是如果有使用不同的货架的需求,那么就需要使用静态工厂方法:

代码片段4 工厂模式演化3_1

```
1 class SampleFactoryMethod {
2     public static synchronized Shelf createShelf(int type) {
3         switch (type) {
4             case 1:
5                 return new SmallShelf();
6             case 2:
7                 return new BigShelf();
8             default:
9                 throw new RuntimeException("没有这种货架");
10        }
11    }
12 }
```

对应的客户端代码如下:

代码片段5 工厂模式演化3_2

```
1 Shelf shelf3 = SampleFactoryMethod.createShelf(1);
2 Shelf shelf4 = SampleFactoryMethod.createShelf(2);
3 Shelf shelf5 = SampleFactoryMethod.createShelf(3);
```

上述使用的都是简单工厂的方法,同样的需求,使用工厂方法也可以实现:

代码片段6 工厂模式演化4_1

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 货架抽象类
5  */
6 public abstract class Shelf {
7
8     /**
9      * 货架容纳货物的最大宽度
10     * 以厘米做单位
11     */
12     private int maxWidth;
13
14     /**
15      * 货架名称
16      */
17     private String shelfName;
18
19     public String getShelfName() {
20         return shelfName;
```

```

21     }
22
23     public void setShelfName(String shelfName) {
24         this.shelfName = shelfName;
25     }
26
27     public int getMaxWidth() {
28         return maxWidth;
29     }
30
31     public void setMaxWidth(int maxWidth) {
32         this.maxWidth = maxWidth;
33     }
34
35     public void put(Goods goods) {
36         System.out.println("将 " + goods.getName() + " 放入 "
37             + this.getShelfName());
38     }
39 }
40
41
42 package cn.steven.pattern.demo.factorymethod;
43
44 /**
45  * 货架抽象类
46  */
47 public abstract class Shelf {
48
49     /**
50      * 货架容纳货物的最大宽度
51      * 以厘米做单位
52      */
53     private int maxWidth;
54
55     /**
56      * 货架名称
57      */
58     private String shelfName;
59
60     public String getShelfName() {
61         return shelfName;
62     }
63
64     public void setShelfName(String shelfName) {
65         this.shelfName = shelfName;
66     }
67
68     public int getMaxWidth() {
69         return maxWidth;
70     }
71

```



```
72 public void setMaxWidth(int maxWidth) {
73     this.maxWidth = maxWidth;
74 }
75
76 public void put(Goods goods) {
77     System.out.println("将 " + goods.getName() + " 放入 "
78         + this.getShelfName());
79 }
80 }
```

当然，还要配合工厂类使用：

代码片段7 工厂模式演化4_2

```
1 package cn.steven.pattern.demo.factorymethod;
2
3 /**
4  * 货架抽象类
5  */
6 public abstract class ShelfFactory {
7
8     /**
9      * 所生产的货架容纳货物的最大宽度
10     * 以厘米做单位
11     */
12     private int maxWidth;
13
14     public int getMaxWidth() {
15         return maxWidth;
16     }
17
18     public void setMaxWidth(int maxWidth) {
19         this.maxWidth = maxWidth;
20     }
21
22     /**
23      * 抽象生产方法
24      * @return 具体的货架对象
25      */
26     abstract public Shelf createShelf();
27
28 }
29
30 package cn.steven.pattern.demo.factorymethod;
31
32 /**
33  * 一种具体的工厂类
34  */
35 public class SmallShelfFactory extends ShelfFactory {
36
37     public SmallShelfFactory() {
38         super();
39     }
40
41     public Shelf createShelf() {
42         return new SmallShelf();
43     }
44
45 }
```


抽象工厂模式就是解决多产品族问题的典型模式。

5.2 抽象工厂模式的结构

抽象工厂模式增加了软件的抽象层次用以适应使用多种工厂来生产多种产品。

5.2.1 抽象工厂模式

抽象工厂模式具备的一个独特的特点就是它支持的工厂具备生产多种产品族的能力，图5-4所示的是一个抽象工厂概念模型图。

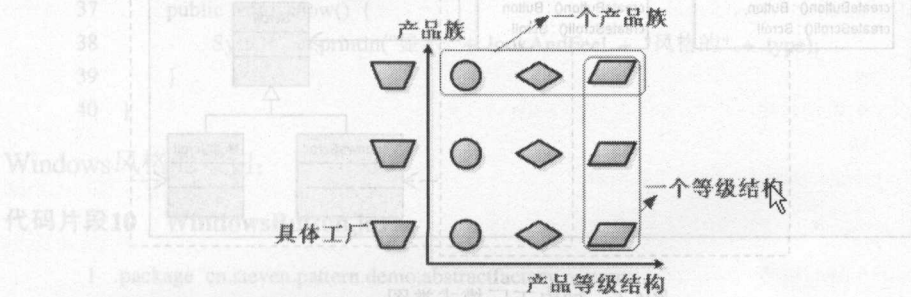


图5-4 抽象工厂概念模型

注意，工厂方法模式只能创造出来一类产品，而需要多类产品时就无能为力了，这就是抽象工厂模式和工厂方法模式最大的区别。

为了说明抽象工厂的设计方法，下面采用一个可以跨平台¹的具有图形用户界面²的软件来演示其结构，这个软件运行在Windows操作系统上时，外观会呈现Windows组件的样子，但是运行在UNIX操作系统上时，会呈现Motif组件的样子，其上的按钮、菜单、滚动条、列表框等组件虽然外观会不一样，但是功能是相同的。

采用抽象工厂的设计理念可以设计出如图5-5所示的类图。图5-5中的参与者的角色如下：

- 抽象工厂（Abstract Factory）角色：在类图中为ComponentFactory抽象类，担任这个角色的是工厂方法模式的核心，它是与应用系统商业逻辑无关的。
- 具体工厂（Concrete Factory）角色：在类图中为WindowsComponentFactory和MotifComponentFactory类，这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。
- 抽象产品（Abstract Product）角色：在类图中为Button和Scroll抽象类，担任这个角色的是工厂方法模式所创建的对象之父类，或它们共同拥有的接口。
- 具体产品（Concrete Product）角色：在类图中为WindowsButton、MotifButton、WindowsScroll、MotifScroll类。抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的对象，其内部一定充满了应用系统的商业逻辑。

¹<http://zh.wikipedia.org/wiki/%E8%B7%A8%E5%B9%B3%E5%8F%B0>。
²<http://zh.wikipedia.org/wiki/%E5%9B%BE%E5%BD%A2%E7%94%A8%E6%88%B7%E7%95%8C%E9%9D%A2>。

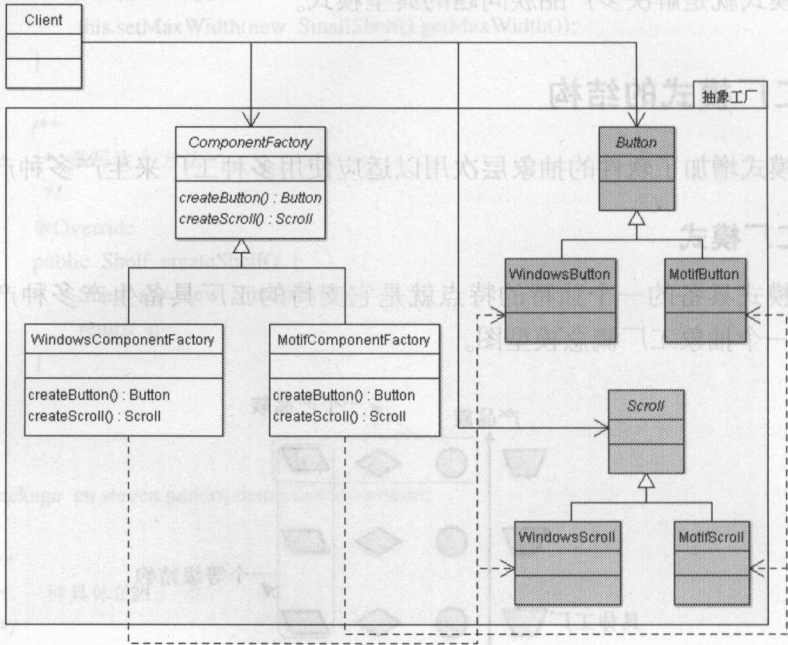


图5-5 抽象工厂模式类图

下面展示的是此模式的代码。

抽象按钮：

代码片段9 Button.java

```
1 package cn.steven.pattern.demo.abstractfactory.pattern;
2
3 /**
4  * 组件抽象类
5  */
6 public abstract class Button {
7
8     /**
9      * 组件名称
10     */
11     private String type = "按钮";
12
13     /**
14      * 视觉感受
15     */
16     private String lookAndFeel;
17
18     public String getLookAndFeel() {
19         return lookAndFeel;
20     }
21
22     public void setLookAndFeel(String lookAndFeel) {
23         this.lookAndFeel = lookAndFeel;
24     }
25 }
```

```

25 public String getType() {
26     return type;
27 }
28
29
30 public void setType(String type) {
31     this.type = type;
32 }
33
34 /**
35  * 显示方法
36  */
37 public void show() {
38     System.out.println("显示" + lookAndFeel + "风格的" + type);
39 }
40 }

```

Windows风格的按钮:

代码片段10 WindowsButton.java

```

1 package cn.steven.pattern.demo.abstractfactory.pattern;
2
3 /**
4  * 具体button实现类
5  */
6 public class WindowsButton extends Button {
7
8     /**
9      * 构造方法
10     */
11     public WindowsButton() {
12         super();
13         this.setLookAndFeel("Windows");
14     }
15 }

```

Motif风格的按钮:

代码片段11 MotifButton.java

```

1 package cn.steven.pattern.demo.abstractfactory.pattern;
2
3 /**
4  * 具体button实现类
5  */
6 public class MotifButton extends Button {
7
8     /**
9      * 构造方法
10     */
11     public MotifButton() {
12         super();

```

```

13         this.setLookAndFeel("Motif");
14     }
15 }

```

抽象滚动条:

代码片段12 Scroll.java

```

1  package cn.steven.pattern.demo.abstractfactory.pattern;
2
3  /**
4   * 组件抽象类
5   */
6  public abstract class Scroll {
7
8      /**
9       * 组件名称
10      */
11      private String type = "滚动条";
12
13      /**
14       * 视觉感受
15       */
16      private String lookAndFeel;
17
18      public String getLookAndFeel() {
19          return lookAndFeel;
20      }
21
22      public void setLookAndFeel(String lookAndFeel) {
23          this.lookAndFeel = lookAndFeel;
24      }
25
26      public String getType() {
27          return type;
28      }
29
30      public void setType(String type) {
31          this.type = type;
32      }
33
34      /**
35       * 显示方法
36       */
37      public void show() {
38          System.out.println("显示" + lookAndFeel + "风格的" + type);
39      }
40 }

```

Windows风格滚动条:

代码片段13 WindowsScroll.java

```
1 package cn.steven.pattern.demo.abstractfactory.pattern;
2
3 /**
4  * 具体Scroll实现类
5  */
6 public class WindowsScroll extends Scroll {
7
8     /**
9      * 构造方法
10     */
11     public WindowsScroll() {
12         super();
13         this.setLookAndFeel("Windows");
14     }
15 }
```

Motif风格滚动条:

代码片段14 MotifScroll.java

```
1 package cn.steven.pattern.demo.abstractfactory.pattern;
2
3 /**
4  * 具体Scroll实现类
5  */
6 public class MotifScroll extends Scroll {
7
8     /**
9      * 构造方法
10     */
11     public MotifScroll() {
12         super();
13         this.setLookAndFeel("Motif");
14     }
15 }
```

抽象工厂抽象类:

代码片段15 ComponentFactory.java

```
1 package cn.steven.pattern.demo.abstractfactory.pattern;
2
3 /**
4  * 抽象工厂的抽象类
5  */
6 public abstract class ComponentFactory {
7
8     /**
9      * 生产一种具体的Button产品
10     */
11     * @return Button对象
```

```

12      */ this.setLookAndFeel("Motif");
13      public abstract Button createButton();
14
15      /**
16       * 生产一种具体的Scroll产品
17       *
18       * @return Scroll对象
19       */
20      public abstract Scroll creaScroll();
21  }

```

Windows控件工厂:

代码片段16 WindowsComponentFactory.java

```

1  package cn.steven.pattern.demo.abstractfactory.pattern;
2
3  /**
4   * 具体抽象工厂
5   */
6  public class WindowsComponentFactory extends ComponentFactory {
7
8      /**
9       * 生产特定系列产品
10     */
11     @Override
12     public Scroll creaScroll() {
13         return new WindowsScroll();
14     }
15
16     /**
17      * 生产特定系列产品
18      */
19     @Override
20     public Button createButton() {
21         return new WindowsButton();
22     }
23
24 }

```

Motif控件工厂:

代码片段17 MotifComponentFactory.java

```

1  package cn.steven.pattern.demo.abstractfactory.pattern;
2
3  /**
4   * 具体抽象工厂
5   */
6  public class MotifComponentFactory extends ComponentFactory {
7
8      /**
9       * 生产特定系列产品

```

```

10     */ void setName();
11     @Override
12     public Scroll creaScroll() {
13         return new MotifScroll();
14     }
15
16     /**
17      * 生产特定系列产品
18      */
19     @Override
20     public Button createButton() {
21         return new MotifButton();
22     }
23
24 }

```

客户端代码:

代码片段18 Client.java

```

1 package cn.steven.pattern.demo.abstractfactory.pattern;
2
3 /**
4  * 客户端代码
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10        /**
11         * 创建变量
12         */
13        ComponentFactory factory;
14        Button button;
15        Scroll scroll;
16
17        /**
18         * 生产Windows风格控件
19         */
20        factory = new WindowsComponentFactory();
21        //下面就可以生产特定的产品了
22        button = factory.createButton();
23        scroll = factory.creaScroll();
24
25        //显示控件
26        button.show();
27        scroll.show();
28
29        /**
30         * 生产Motif风格控件
31         */
32        factory = new MotifComponentFactory();

```



```

33      //下面就可以生产特定的产品了
34      button = factory.createButton();
35      scroll = factory.creaScroll();
36
37      //显示控件
38      button.show();
39      scroll.show();
40
41  }
42  }

```

客户端代码的运行结果如图5-6所示。

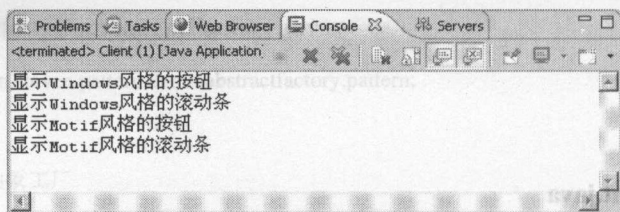


图5-6 Client.java运行结果

由图5-6可见，经过了抽象工厂模式的设计，我们就可以用不同的工厂来生产不同系列的产品，客户端使用的代码也稳定下来了，当有新的系列产品出现时，比如现在要加入一种苹果风格的控件，那么只需要设计一种苹果系列的产品类，然后做出苹果风格的控件工厂即可，客户端原来的代码是无需更改的。

虽然新增一种系列的产品是很容易的，但是请读者思考一下，如果需要增加一种产品类型，是否还是很简单呢？

5.2.2 使用抽象工厂模式解决商品上架问题

经过上一节的学习，对于商品上架问题的新需求：增加商品上架所需的产品类型要求，可以设计为如图5-7所示。

在图5-7中，比起传统的抽象工厂其中多设计了一个类PropTool，此类的作用是读取config.properties配置文件，用以确定商品所需要的特定工厂。

下面展示的是产品族。

货架产品抽象类：

代码片段19 Shelf.java

```

1  package cn.steven.pattern.demo.abstractfactory;
2
3  /**
4   * 货架抽象类
5   */
6  public abstract class Shelf {
7
8      /**
9       * 货架名称
10      */

```

```
11 private String name;
12
13 public String getName() {
14     return name;
15 }
16
17 public void setName(String name) {
18     this.name = name;
19 }
20
21 /**
22  * 存放货物方法
23  */
24 public void put(Goods goods) {
25     System.out.println(goods.getName() + " 放入 " + getName());
26 }
27 }
```

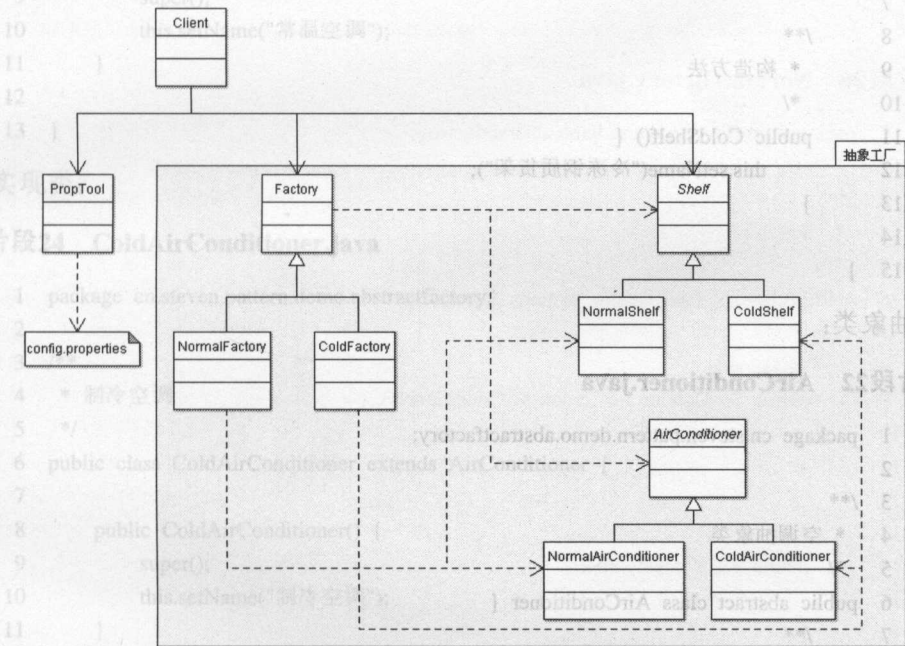


图5-7 抽象工厂解决上架问题类图

货架实现类:

代码片段20 NormalShelf.java

```
1 package cn.steven.pattern.demo.abstractfactory;
2
3 /**
4  * 普通货架
5  */
6 public class NormalShelf extends Shelf {
7
8     /**
```

```
9      * 构造方法
10     */
11     public NormalShelf() {
12         this.setName("普通木质货架");
13     }
14
15 }
```

货架实现类:

代码片段21 ColdShelf.java

```
1 package cn.steven.pattern.demo.abstractfactory;
2
3 /**
4  * 冷冻货架
5  */
6 public class ColdShelf extends Shelf {
7
8     /**
9      * 构造方法
10     */
11     public ColdShelf() {
12         this.setName("冷冻钢质货架");
13     }
14
15 }
```

空调抽象类:

代码片段22 AirConditioner.java

```
1 package cn.steven.pattern.demo.abstractfactory;
2
3 /**
4  * 空调抽象类
5  */
6 public abstract class AirConditioner {
7     /**
8      * 空调名称
9      */
10     private String name;
11
12     public String getName() {
13         return name;
14     }
15
16     public void setName(String name) {
17         this.name = name;
18     }
19
20     /**
21      * 空调启动
```

```
private String name;
11
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21
22
23
24
25
26
27
```



图 5-7 抽象工厂模式工厂类图

类图实现类图

代码片段20 NormalShelf.java

```
1 package cn.steven.pattern.demo.abstractfactory;
2
3 /**
4  * 普通货架
5  */
6 public class NormalShelf extends Shelf {
7
8     /**
```



```
22     */
23     public void turnOn(){
24         System.out.println(getName()+" 开启!");
25     }
26 }
```

空调实现类:

代码片段23 NormalAirConditioner.java

```
1 package cn.steven.pattern.demo.abstractfactory;
2
3 /**
4  * 常温空调
5  */
6 public class NormalAirConditioner extends AirConditioner {
7
8     public NormalAirConditioner() {
9         super();
10        this.setName("常温空调");
11    }
12
13 }
```

空调实现类:

代码片段24 ColdAirConditioner.java

```
1 package cn.steven.pattern.demo.abstractfactory;
2
3 /**
4  * 制冷空调
5  */
6 public class ColdAirConditioner extends AirConditioner {
7
8     public ColdAirConditioner() {
9         super();
10        this.setName("制冷空调");
11    }
12
13 }
```

经过了上述代码的展示,读者可以看出,根据所需温度的不同,我们需要为商品准备不同的物品和条件,对于这种需要多种系列产品的要求,抽象工厂模式是非常合适的,但是在具体的使用过程中还需要根据客户的需求做一些便捷上的考虑。

抽象工厂抽象类:

代码片段25 Factory.java

```
1 package cn.steven.pattern.demo.abstractfactory;
2
3 /**
4  * 抽象工厂抽象类
```

```
5  */
6  public abstract class Factory {
7      public NormalShelf() {
8          /**
9           * 生产具体的货架对象
10          *
11          * @return 具体货架
12          */
13      public abstract Shelf createShelf();
14
15      /**
16       * 生产具体的空调对象
17       *
18       * @return 具体空调
19       */
20      public abstract AirConditioner createAirConditioner();
21  }
```

具体工厂：

代码片段26 NormalFactory.java

```
1  package cn.steven.pattern.demo.abstractfactory;
2
3  /**
4   * 常温工厂
5   */
6  public class NormalFactory extends Factory {
7
8      @Override
9      public AirConditioner createAirConditioner() {
10         return new NormalAirConditioner();
11     }
12
13     @Override
14     public Shelf createShelf() {
15         return new NormalShelf();
16     }
17
18 }
```

具体工厂：

代码片段27 ColdFactory.java

```
1  package cn.steven.pattern.demo.abstractfactory;
2
3  /**
4   * 冷冻工厂
5   */
6  public class ColdFactory extends Factory {
7
8      @Override
```

```

9      public AirConditioner createAirConditioner() {
10         return new ColdAirConditioner();
11     }
12
13     @Override
14     public Shelf createShelf() {
15         return new ColdShelf();
16     }
17
18 }

```

通过代码片段25、代码片段26和代码片段27这三段代码可以看出，现在已经可以通过抽象工厂来生产不同系列的产品了，但问题是客户还需要明确地知道要使用哪一种具体的工厂实现类，为了解决这个客户易用性的问题，我们首先可以把所有的可以用来判别使用哪种抽象工厂的情况放置在一个枚举¹中：

代码片段28 GoodsType.java

```

1  package cn.steven.pattern.demo.abstractfactory;
2
3  /**
4   * 货物类型的枚举
5   */
6  public enum GoodsType {
7      /**
8       * 常温类型
9       */
10     normalTemperature,
11
12     /**
13      * 冷冻类型
14      */
15     coldTemperature
16 }

```

商品类中需要制定这种枚举值：

代码片段29 Goods.java

```

1  package cn.steven.pattern.demo.abstractfactory;
2
3  /**
4   * 商品类
5   */
6  public class Goods {
7
8      /**
9       * 商品名
10      */
11     private String name;

```

¹<http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html>.


```

12  /**
13  * 商品类别
14  */
15  private GoodsType goodsType;
16
17  public String getName() {
18      return name;
19  }
20  public void setName(String name) {
21      this.name = name;
22  }
23  public GoodsType getGoodsType() {
24      return goodsType;
25  }
26  public void setGoodsType(GoodsType goodsType) {
27      this.goodsType = goodsType;
28  }
29
30  /**
31  * 构造方法
32  */
33  public Goods(String name, GoodsType goodsType) {
34      super();
35      this.name = name;
36      this.goodsType = goodsType;
37  }
38
39  }

```

下面展示的是如何把特定的枚举值和其所需的具体工厂关联起来的代码，首先看一下配置文件是如何声明其关联关系的：

代码片段30 config.properties

```

1  #工厂配置文件
2  normalTemperature=cn.steven.pattern.demo.abstractfactory.NormalFactory
3  coldTemperature=cn.steven.pattern.demo.abstractfactory.ColdFactory

```

配置文件的解析助手类：

代码片段31 PropTool.java

```

1  package cn.steven.pattern.demo.abstractfactory;
2
3  import java.io.IOException;
4  import java.util.Properties;
5
6  /**
7   * 解析properties文件的工具类
8   */
9  public class PropTool {
10
11      private static Properties prop;

```

```
12 private static final String filePath =  
13     "/cn/steven/pattern/demo/abstractfactory/config.properties";  
14 package cn.steven.pattern.demo.abstractfactory;  
15 public synchronized static String getProperty(String key) {  
16     if (prop == null) {  
17         /**  
18          * 创建配置对象  
19          */  
20         prop = new Properties();  
21         try {  
22             /**  
23              * 解析文件  
24              */  
25             prop.load(PropTool.class  
26                 .getResourceAsStream(filePath));  
27         } catch (IOException e) {  
28             System.out.println(filePath+" 解析失败!");  
29         }  
30     }  
31     /**  
32     * 返回找到的值  
33     */  
34     return prop.getProperty(key);  
35 }  
36 }
```

至此,功能性的代码已经完成了,最后看一下客户端是如何使用这些代码的:

代码片段32 Client.java

```
1 package cn.steven.pattern.demo.abstractfactory;  
2  
3 /**  
4  * 客户端代码  
5  */  
6 public class Client {  
7  
8     public static void main(String[] args) {  
9  
10         /**  
11          * 创建一个需要常温保存的货物  
12          */  
13         Goods goods = new Goods("餐具", GoodsType.normalTemperature);  
14         /**  
15          * 获得合适的工厂  
16          */  
17         Factory factory = AbstractFactoryHelper.getFactory(goods);  
18         /**  
19          * 生产对象并操作  
20          */  
21         AirConditioner airConditioner = factory  
22             .createAirConditioner();
```

```
23         airConditioner.turnOn();
24
25         Shelf shelf = factory.createShelf();
26         shelf.put(goods);
27
28         System.out.println();
29
30         /**
31          * 创建一个需要冷冻保存的货物
32          */
33         Goods goods1 = new Goods("冰激凌", GoodsType.coldTemperature);
34         /**
35          * 获得合适的工厂
36          */
37         Factory factory1 = AbstractFactoryHelper.getFactory(goods1);
38         /**
39          * 生产对象并操作
40          */
41         AirConditioner airConditioner1 = factory1
42             .createAirConditioner();
43         airConditioner1.turnOn();
44
45         Shelf shelf1 = factory1.createShelf();
46         shelf1.put(goods);
47     }
48
49 }
```

在客户端代码中，使用了抽象工厂的助手类，所以避免了直接与特定的工厂实现耦合，同时简化了客户端的代码。

运行结果如图5-8所示。

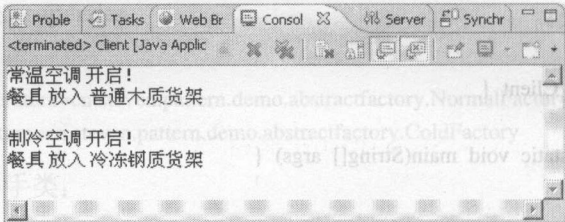


图5-8 Client.java运行结果

由图5-8的运行结果可见根据不同的商品可以生产出匹配其需要的一系列产品，并且客户端的代码是很简单的。

5.2.3 抽象工厂模式在JDK中的实例

抽象工厂模式在JDK的源代码中的使用也是非常常见的，本章起始部分的例子就充分说明了图形界面是适合使用抽象工厂模式的。

下面我们通过查看使用图形界面的方法了解一下JDK是如何使用此模式的。下面的代码功能实现了一个窗口，其中的视感使用的是MetalLookAndFeel风格：

代码片段33 SwingTest.java

```
1 package cn.steven.pattern.demo.abstractfactory.jdk;
2
3 import java.awt.FlowLayout;
4 import javax.swing.JButton;
5 import javax.swing.JFrame;
6 import javax.swing.JTextField;
7 import javax.swing.UIManager;
8 import javax.swing.UnsupportedLookAndFeelException;
9
10 /**
11  * Swing图形化演示
12  */
13 public class SwingTest {
14
15     public static void main(String[] args)
16         throws ClassNotFoundException, InstantiationException,
17         IllegalAccessException, UnsupportedLookAndFeelException {
18
19         /**
20          * 设置视感，默认提供以下几个
21          * javax.swing.plaf.metal.MetalLookAndFeel
22          * com.sun.java.swing.plaf.motif.MotifLookAndFeel
23          * com.sun.java.swing.plaf.windows.WindowsLookAndFeel
24          */
25
26         String lf = "javax.swing.plaf.metal.MetalLookAndFeel";
27         /**
28          * 设置视感
29          */
30         UIManager.setLookAndFeel(lf);
31
32         /**
33          * 创建窗口
34          */
35         JFrame window = new JFrame("swing测试");
36         /**
37          * 设置面板布局为流式布局
38          */
39         window.getContentPane().setLayout(new FlowLayout());
40
41         /**
42          * 创建控件
43          */
44         JButton button = new JButton("按钮");
45         JTextField textField = new JTextField("测试文本");
46
47         /**
48          * 设置窗口关闭方法
49          */
```

```
50 window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51
52 /**
53  * 组装控件
54  */
55 window.getContentPane().add(button);
56 window.getContentPane().add(textField);
57
58 /**
59  * 窗口定位及显示
60  */
61 window.setLocationRelativeTo(null);
62 window.pack();
63 window.setVisible(true);
64
65 System.out.println("视感类: "
66     + UIManager.getLookAndFeel().getClass());
67 System.out.println("按钮: " + button.getUI().getClass());
68 System.out.println("文本: " + textField.getUI().getClass());
69 }
70 }
```

运行结果如图5-9所示。



图5-9 MetalLookAndFeel风格

将代码片段33中第26行代码改为

```
26 String lf = com.sun.java.swing.plaf.motif.MotifLookAndFeel;
```

则结果如图5-10所示。



图5-10 MotifLookAndFeel风格

由此可见，更换了一种LookAndFeel则相当于使用了一种抽象工厂，此工厂可以创建一系列外观相似的控件，在运行时刻就可以显示出特定的外观了。

下面看一下JDK提供的一种叫做Service Provider¹的机制，此机制的作用是可以配置文件的方式通知JVM某个服务的默认实现类是什么。

具体做法是在源代码的META-INF/services目录中存放一些文件，这些文件的文件名是抽象的类或接口，文件的内容以文本的方式写出实现类的名称。

下面以创建DocumentBuilderFactory工厂为例：

代码片段34 ServiceProvider.java

```
1 package cn.steven.pattern.demo.abstractfactory.jdk;
2
3 import javax.xml.parsers.DocumentBuilderFactory;
4
5 /**
6  * Service Provider
7  */
8 public class ServiceProvider {
9     public static void main(String[] args) {
10         DocumentBuilderFactory factory = DocumentBuilderFactory
11             .newInstance();
12         System.out.println(factory.getClass());
13     }
14 }
```

从DocumentBuilderFactory中可以看出，此处设计使用了某种工厂模式，但是从代码片段34的第10行看不出具体实例化的是哪一个具体工厂类，下面先来看一下运行结果，如图5-11所示。

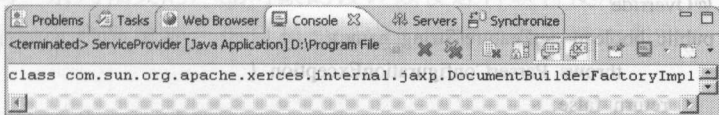


图5-11 ServiceProvider运行结果1

很奇怪，创建出来的是一个没有在代码中出现的类，来看一下DocumentBuilderFactory中的代码：

代码片段35 DocumentBuilderFactory.java片段

```
121 public static DocumentBuilderFactory newInstance() {
122     try {
123         return (DocumentBuilderFactory) FactoryFinder.find(
124             /* The default property name according to the JAXP spec */
125             "javax.xml.parsers.DocumentBuilderFactory",
126             /* The fallback implementation class name */
127             "com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl");
128     } catch (FactoryFinder.ConfigurationError e) {
129         throw new FactoryConfigurationError(e.getException(),
130             e.getMessage());
131     }
132 }
133 }
```

¹<http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html>.

FactoryFinder就是提供Service Provider机制的工具类，此段代码的意思就是使用Service Provider机制查找DocumentBuilderFactory的实现类，如果找不到的话，再使用一个默认的实现类。明白了此机制之后，我们如果想要自己实现此类而不用JDK提供的默认类，那么方法就分为以下几步：

第一步：实现DocumentBuilderFactory抽象类。

代码片段36 MyDocumentBuilderFactory.java

```

1 package cn.steven.pattern.demo.abstractfactory.jdk;
2
3 import javax.xml.parsers.DocumentBuilder;
4 import javax.xml.parsers.DocumentBuilderFactory;
5 import javax.xml.parsers.ParserConfigurationException;
6
7 /**
8  * 自己的DocumentBuilderFactory类
9  */
10 public class MyDocumentBuilderFactory extends DocumentBuilderFactory {
11
12     @Override
13     public Object getAttribute(String name)
14         throws IllegalArgumentException {
15         return null;
16     }
17
18     @Override
19     public boolean getFeature(String name)
20         throws ParserConfigurationException {
21         return false;
22     }
23
24     @Override
25     public DocumentBuilder newDocumentBuilder()
26         throws ParserConfigurationException {
27         return null;
28     }
29
30     @Override
31     public void setAttribute(String name, Object value)
32         throws IllegalArgumentException {
33
34     }
35
36     @Override
37     public void setFeature(String name, boolean value)
38         throws ParserConfigurationException {
39
40     }
41
42 }

```

第二步：在源代码目录中创建META-INF/services文件夹。

第三步：在此目录中创建一个下述文本文件。

代码片段37 javax.xml.parsers.DocumentBuilderFactory

```
1 cn.steven.pattern.demo.abstractfactory.jdk.MyDocumentBuilderFactory
```

现在再来运行代码片段34，得到的结果如图5-12所示。

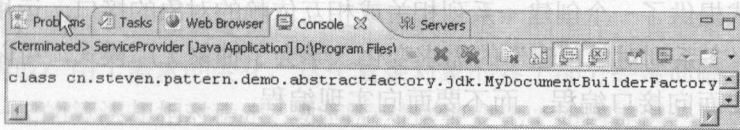


图5-12 ServiceProvider运行结果2

可见，这样的设计就可以通过一种简单的方式将客户代码所使用的工厂和真正的工厂创建完全解耦，达到设计上的优美性。

如果要创建的工厂实现类的种类复杂，建议最好还是使用外部XML配置文件进行配置，或使用IoC¹的思想进行注入处理。

5.2.4 抽象工厂模式的使用范围

在以下情况下可以使用Abstract Factory模式：

- 一个系统要独立于它的产品的创建、组合和表示时。
- 一个系统要由多个产品系列中的一个来配置时。
- 当你要强调一系列相关的产品对象的设计以便进行联合使用时。
- 当你提供了一个产品类库，但只想显示它们的接口而不是实现时。

优点：

- 具体产品从客户代码中被分离出来。
- 容易改变产品的系列。
- 将一个系列的产品族统一到一起创建。

缺点：

- 在产品族中扩展新的产品是很困难的，它需要修改抽象工厂的接口。
- 使软件结构更加复杂。

5.2.5 与其他模式的关系

抽象工厂模式与工厂方法模式的区别：

• 工厂方法模式是一种极端情况的抽象工厂模式，而抽象工厂模式可以看成是工厂方法模式的一种推广。

• 工厂方法模式是用来创建一个产品的等级结构的，而抽象工厂模式是用来创建多个产品的等级结构的。工厂方法创建一般只有一个方法，创建一种产品。抽象工厂一般有多个方法，创建一系列产品。

• 工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个抽象产品类。工厂方法模式

¹<http://www.jdon.com/AOPdesign/loc.htm>.

的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个。
工厂的实现通常使用Singleton模式。一个应用中一般每个产品系列只需使用一个具体工厂的实例，因此，工厂通常最好实现为一个Singleton模式。

5.3 抽象工厂模式总结

抽象工厂模式提供了一个创建一系列相关或相互依赖的对象的接口，运用抽象工厂模式的关键点在于应对“多系列对象创建”的需求变化。学会了抽象工厂模式，可以很好地理解面向对象中的原则：要面向接口编程，而不要面向实现编程。

图 5-12 26715617rvider 类图

```
1 import javax.xml.parsers.*;
2 import javax.xml.parsers.*;
3 import javax.xml.parsers.*;
```

4 import javax.xml.parsers.*;

5 /**

6 * 自己的DocumentBuilder

7 */

8 public class MyDocumentBuilder extends DocumentBuilder {

9

10 @Override

11 public Object getAttribute(String name, Object value)

12 throws ParserConfigurationException {

13 return null;

14 }

15

16 throws ParserConfigurationException {

17 return false;

18 }

19

20 @Override

21 public DocumentBuilder newDocumentBuilder()

22 throws ParserConfigurationException {

23 }

24 }

25

26 @Override

27 public void setAttribute(String name, Object value)

28 throws IllegalArgumentException {

29 }

30 }

31

32

33

34

35

36

37

38

39

40

第6章 建造者模式 (Builder Factory)

建造者模式将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示¹。

在软件系统中，有时候面临着要创建“一个复杂对象”的工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。如何应对这种变化？如何提供一种“封装机制”来隔离出“复杂对象的各个部分”的变化，从而保持系统中的“稳定构建算法”不随着需求改变而改变？这就是建造者模式可以解决的。

在现实生活中也有此类案例，比如在某快餐店点餐，点餐的过程相当于“一个复杂对象”的创建工作，比如需要点一个儿童套餐，此套餐由主食、副食、饮料和一种玩具构成，其中的每一个部分都可能变化，收银员会将需要的每一种物品都告诉后厨服务员进行“构建”，收银员就成为一个“指导者”角色，他不清楚每一种食品是怎样做出来的，但是他知道需要做什么东西。而“后厨”充当了“建造者”的角色，由他来生产物品。

在生产企业中建造者模式会更加常见，比如生产一种汽车，需要经过很多的零件制作过程，最后才能完成整部车的装配，流水线充当“指导者”，它会以某种方式构建产品，而工人的集合就是一种“建造者”，他们各司其职，共同完成工作。

6.1 商品捆绑销售的问题

超市中为了促进商品销售，通常会进行商品的捆绑销售。比如顾客买一个大号的中华牙膏，将会附送一个小号的中华牙膏和一个牙刷。这三样物品是一起售卖的，不可以单卖。再比如顾客买了一台电视，将会附送多件物品，比如耳机、麦克、光盘、线材等。

图6-1所示的是一个真实的捆绑销售图片，由此图可见捆绑的种类可能多种多样，最重要的是这些组合的种类要可以轻易变化。



图6-1 捆绑销售

¹Separate the construction of a complex object from its representation so that the same construction process can create different representations.-GOF[95]

由此可见，捆绑销售的物品需要按照物品的种类进行搭配，不同的物品将搭配不同的捆绑物品，搭配的种类很多，这种销售方法的特点是：被搭配销售的主物品不允许单独销售，但是搭配的物品可以单独销售。

根据此法则，一种初步的解决方法可能由如下代码构成：

代码片段1 Step_1.java

```

1 package cn.steven.pattern.demo.builder.quest;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 捆绑销售的初步解决方案
8  */
9 public class Step_1 {
10     public static void main(String[] args) {
11         // 捆绑销售货物列表
12         List goodsList = new ArrayList();
13         // 主商品
14         goodsList.add(new Goods("PSP主机"));
15         //捆绑物品
16         goodsList.add(new Goods("HIFI耳机"));
17         goodsList.add(new Goods("4G存储卡"));
18     }
19 }
20
21 /**
22  * 商品类
23  */
24 class Goods {
25     /**
26      * 商品名
27      */
28     private String name;
29
30     public String getName() {
31         return name;
32     }
33
34     public void setName(String name) {
35         this.name = name;
36     }
37
38     /**
39      * 构造方法
40      * @param name
41      */
42     public Goods(String name) {
43         super();
44         this.name = name;

```

45 }

46 }

由以上代码中可见，具体的商品组装代码是代码片段1中第14行~第17行，其原理就是直接将几种商品放置于一个集合中，但是这样做有以下几个缺点：

- 商品的构建过程耦合于客户端代码中，如果未来需要对商品的构建过程加以统一处理，将要修改大量的客户端代码。
- 商品的组合被耦合于客户端代码中，未来如果将改变商品组合，会修改大量客户端代码。
- 增加新的商品以及其组合都变得困难。
- 客户端需要得知所有的组合，难以改变组合。

经过上一章的学习，我们已经可以理解抽象工厂的设计和使用方法。看到了上面的问题解决方案后，有的读者可能就会提出这样的问题：

“既然需要创建的产品是变化的，那么何不将产品的创建交给抽象工厂生产，这样前端就可以直接使用工厂生产的产品，无需耦合特定的产品了。”

为了更加清晰地分析这个需求，下面使用类图和代码的方式来研究一下。

根据需求可以绘制类图如图6-2所示。

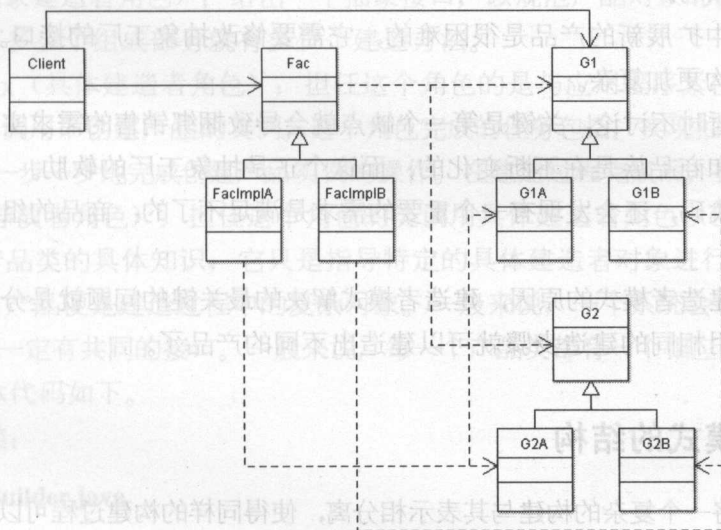


图6-2 使用抽象工厂的类图

使用此结构的代码可以如下：

代码片段2 Step_2.java

```
1 package cn.steven.pattern.demo.builder.quest;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 抽象工厂示意代码
8  */
9 public class Step_2 {
```



```
10 public static void main(String[] args) {
11     // 捆绑销售货物列表
12     List goodsList = new ArrayList();
13     //组合A
14     Fac fac = new FacImplA();
15     goodsList.add(fac.createG1());
16     goodsList.add(fac.createG2());
17
18     //清空列表
19     goodsList.clear();
20     //组合B
21     Fac fac = new FacImplB();
22     goodsList.add(fac.createG1());
23     goodsList.add(fac.createG2());
24 }
25 }
```

注意代码片段2中提到的第一种捆绑销售物品的代码为第12行~第16行，它的原理是分离了创建过程和创建代码，现在只需要使用特定的工厂即可完成特定商品的创建，但是不要忘记，抽象工厂是有缺点的：

- 在产品族中扩展新的产品是很困难的，它需要修改抽象工厂的接口。
- 使软件结构更加复杂。

第二个缺点暂时不讨论，关键是第一个缺点就会导致捆绑销售的需求实现起来极为困难。因为商品的组合和商品族是在不断变化的，而这个正是抽象工厂的软肋。

对照类图和代码，还会发现有一个重要的需求是满足不了的：商品的组合情况需要方便地变化。

这就是引入建造者模式的原因，建造者模式解决的最关键的问题就是分离建造过程和建造步骤，这样，使用相同的建造步骤就可以建造出不同的产品了。

6.2 建造者模式的结构

建造者模式将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

6.2.1 建造者模式

为了说明建造者模式的设计方法，下面使用一个快餐店的点餐过程作为例子进行讲解。在这个环境中顾客通常会单个点食品，这时软件的实现只需使用简单工厂即可，但是如果点的是套餐，就比较麻烦了，比如一个KFC的套餐，如图6-3所示。

套餐的特点是：

- 每种套餐的食品大类一样：主食、辅食、饮料、赠送品。
- 套餐中每种食品（任意一种）都不一样。
- 这种组合是异变的。

通过面向对象的设计原则可知，如果遇到变化的和不变化的，则将变化进行抽象和分离。在以上点餐的案例中，可以分析出变化分别为：

- 配餐的组合过程。

• 生成配餐的具体食品。

使用建造者模式的类图如图6-4所示。



图6-3 套餐示例

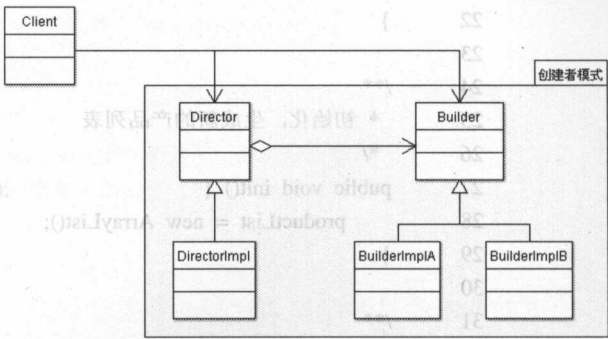


图6-4 建造者模式类图

在图6-4中的各种参与者为：

- **Builder**（抽象建造者角色）：给出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，有多少个组成部分就有多少个建造方法。
- **BuilderImplx**（具体建造者角色）：担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序的调用下创建产品的实例。这个角色完成的任务包括：实现抽象建造者**Builder**所声明的接口，给出一步一步地完成创建产品实例的操作。在建造过程完成后，提供产品的实例。
- **Director**（导演者角色）：担任这个角色的类调用具体建造者角色以创建产品对象。导演者角色并没有产品类的具体知识，它只是指导特定的具体建造者对象进行创建的过程。
- **产品角色**：产品便是建造过程中的复杂对象。一般来说，一个系统会有多个产品类，而且这些产品类并不一定有共同的接口。一般来说，每一个产品类都有一个相应的具体建造者类。

本示例的具体代码如下。

抽象建造者类：

代码片段3 Builder.java

```
1 package cn.steven.pattern.demo.builder.pattern;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 抽象创建者
8  */
9 abstract public class Builder {
10
11     /**
12      * 商品的列表
13      */
14     private List productList;
15 }
```

```
16 public List getProductList() {
17     return productList;
18 }
19
20 public void setProductList(List productList) {
21     this.productList = productList;
22 }
23
24 /**
25  * 初始化，生成新的产品列表
26  */
27 public void init() {
28     productList = new ArrayList();
29 }
30
31 /**
32  * 以下方法为创建商品部分的方法
33  * 注意此处不声明为抽象方法的好处是：
34  * 如果具体子类没有某个创建方法，则会自动调用父类的空方法，
35  * 而抽象方法则强制子类实现，造成编程上的冗余
36  */
37 public void createFoodA() { }
38     //空方法
39 }
40
41 public void createFoodB() { }
42     //空方法
43 }
44
45 public void createDrink() { }
46     //空方法
47 }
48
49 public void createAddition() { }
50     //空方法
51 }
52
53 }
```

要注意一个设计建造者模式的要点：在代码片段3中第37行~第51行声明了很多空方法，而不是声明为抽象方法，这种设计给了其子类更大的灵活性。

儿童套餐实现类：

代码片段4 BuilderImplA.java

```
1 package cn.steven.pattern.demo.builder.pattern;
2
3 /**
4  * 儿童套餐
5  */
6 public class BuilderImplA extends Builder {
7
```




```

8      /**
9       * 创建主食
10      */
11     public void createFoodA() {
12         getProductList().add("麦香鱼汉堡");
13     }
14
15     /**
16      * 创建辅食
17      */
18     public void createFoodB() {
19         getProductList().add("小薯条");
20     }
21
22     /**
23      * 创建饮料
24      */
25     public void createDrink() {
26         getProductList().add("奶昔");
27     }
28
29     /** 构造方法
30      * 创建附加物
31      */
32     public void createAddition() {
33         getProductList().add("喜洋洋玩具");
34     }
35 }

```

成人套餐实现类:

代码片段5 BuilderImplB.java

```

1  package cn.steven.pattern.demo.builder.pattern;
2
3  /**
4   * 成人套餐
5   */
6  public class BuilderImplB extends Builder {
7
8      /**
9       * 创建主食
10      */
11     public void createFoodA() {
12         getProductList().add("巨无霸汉堡");
13     }
14
15     /**
16      * 创建辅食
17      */
18     public void createFoodB() {
19         getProductList().add("大薯条");

```

```
20     }
21     return productList;
22     /**
23      * 创建饮料
24      */
25     public void createDrink() {
26         getProductList().add("大杯可乐");
27     }
28     /**
29      * 没有附加物，所以没有定义createAddition方法
30      * 当调用此方法时，会使用父类的方法
31      */
32     // productList = new ArrayList();
33 }
```

下面展示的是控制建造过程的抽象导演类：

代码片段6 Director.java

```
1 package cn.steven.pattern.demo.builder.pattern;
2
3 import java.util.List;
4
5 /**
6  * 抽象导演类
7  */
8 abstract public class Director {
9
10     /**
11      * 聚合的一个具体创建对象
12      */
13     private Builder builder;
14
15
16     public Builder getBuilder() {
17         return builder;
18     }
19
20     public void setBuilder(Builder builder) {
21         this.builder = builder;
22     }
23
24     /**
25      * 构造方法
26      * @param builder
27      *      传入的具体创建对象
28      */
29
30     public Director(Builder builder) {
31         super();
32         this.builder = builder;
33     }
```

```
34
35 /** * 显示结果
36 * 构建产品
37 */ Client.viewList(bList);
38 public abstract List construct();
39
40 }
```

导演类的实现类:

代码片段7 DirectorImpl.java

```
1 package cn.steven.pattern.demo.builder.pattern;
2
3 import java.util.List;
4
5 /**
6 * 具体导演类
7 */
8 public class DirectorImpl extends Director {
9
10     /**
11     * 构造方法
12     *
13     * @param builder
14     *     传入的具体创建对象
15     */
16     public DirectorImpl(Builder builder) {
17         super(builder);
18     }
19
20     /**
21     * 重写父类的构建方法
22     */
23     @Override
24     public List construct() {
25         /**
26         * 初始化创建对象
27         */
28         getBuilder().init();
29
30         /**
31         * 创建过程
32         */
33         getBuilder().createFoodA();
34         getBuilder().createFoodB();
35         getBuilder().createDrink();
36         getBuilder().createAddition();
37
38         /**
39         * 得到产品
40         */
41     }
```



```
41         return getBuilder().getProductList();
42
43     }
44
45 }
```

由代码片段7中的construct方法可以看出，这个类控制着产品的构建过程，但是它并不知道构建的具体细节，在构建结束后才会返回产品集合。

客户端代码：

代码片段8 Client.java

```
1 package cn.steven.pattern.demo.builder.pattern;
2
3 import java.util.List;
4
5 /**
6  * 客户端代码
7  */
8 public class Client {
9     public static void main(String[] args) {
10
11         /**
12          * 声明一个导演变量
13          */
14         Director director;
15
16         /**
17          * 构建儿童套餐的过程
18          * 创建导演，传入特定的建造对象
19          */
20         director = new DirectorImpl(new BuilderImplA());
21
22         /**
23          * 建造
24          */
25         List aList = director.construct();
26
27         /**
28          * 显示结果
29          */
30         Client.viewList(aList);
31
32         /**
33          * 构建成人套餐的过程
34          * 创建导演，传入特定的建造对象
35          */
36         director = new DirectorImpl(new BuilderImplB());
37
38         /**
39          * 建造
40          */
41         List bList = director.construct();
42     }
43 }
```

```
40      /**
41       * 显示结果
42       */
43      Client.viewList(bList);
44  }
45  }
46
47  /**
48   * 打印列表信息的工具方法
49   *
50   * @param list
51   */
52  public static void viewList(List list) {
53      for (Object object : list) {
54          System.out.print(object + " ");
55      }
56      System.out.println();
57  }
58  }
```

由代码片段8中第20行~第43行可以看出，创建产品的过程十分简单，创建不同产品的过程也是完全相同的，如此一来，客户端的代码就具备了稳定性。当有不同的套餐出现时，需要做的就是创建一个具体的Builder实现类，Director类和客户端代码是不需要更改的。

运行结果如图6-5所示。



图6-5 运行结果

由结果可见，不同的套餐的构建过程是完全一样的，但是由于使用了不同的创建类，结果又完全不同，这样就完全解决了套餐的问题。

6.2.2 使用建造者模式解决捆绑销售问题

经过了上一节的学习，下面就可以使用建造者模式来具体解决捆绑销售的问题了。在本节中，除了要解决捆绑销售的简单问题，还将引入不同的商品销售配置版，这种需求在销售时也经常见到，比如标准版的PSP包括主机、充电器、耳机，但是豪华版的PSP不只包含标准版的所有配件，可能还会增加贴膜、硅胶套、记忆棒、游戏光盘等。

根据需求可以设计出如图6-6所示的类图。

注意图6-4和图6-6的主要区别在于后者增加了一个产品类，和一个导演实现类，结构更加复杂。

下面先来看一下产品类：

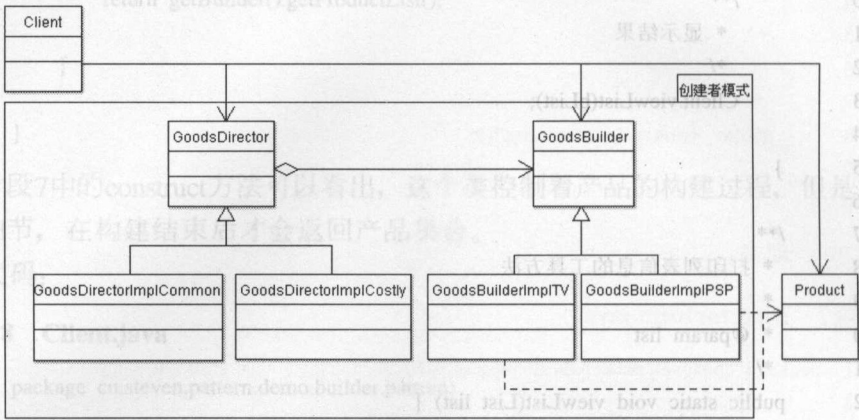


图6-6 使用建造者模式解决捆绑销售的类图

代码片段9 Product.java

```
1 package cn.steven.pattern.demo.builder.quest;
2 public class Client {
3     /**
4      * 产品类
5      */
6     public class Product {
7
8         /**
9          * 产品名称
10          */
11         private String name;
12
13         /**
14          * 产品价格
15          */
16         private float price;
17
18         public String getName() {
19             return name;
20         }
21
22         public void setName(String name) {
23             this.name = name;
24         }
25
26         public float getPrice() {
27             return price;
28         }
29
30         public void setPrice(float price) {
31             this.price = price;
32         }
33
34         /**
```



```

35     * 构造方法 createAdd2Product() {
36     *     getProductList().add(new Product("蓝牙耳机", 199.05f));
37     * @param name
38     * @param price
39     */
40     public Product(String name, float price) {
41         super();
42         this.name = name;
43         this.price = price;
44     }
45
46     /**
47     * 重写Object的toString方法
48     */
49     @Override
50     public String toString() {
51         return "(" + this.getName() + "/" + this.getPrice() + ")";
52     }
53     public void createMainProduct() {

```

抽象建造类:

代码片段10 GoodsBuilder.java

```

1 package cn.steven.pattern.demo.builder.quest;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 抽象商品建造类
8  */
9 public abstract class GoodsBuilder {
10
11     /**
12     * 商品的列表
13     */
14     private List<Product> productList;
15
16     public List<Product> getProductList() {
17         return productList;
18     }
19
20     public void setProductList(List<Product> productList) {
21         this.productList = productList;
22     }
23
24     /**
25     * 初始化, 生成新的产品列表
26     */
27     public void init() {
28         productList = new ArrayList<Product>();

```

```
29     }
30
31     /**
32      * 以下方法为创建商品部分的方法
33      * 注意此处不声明为抽象方法的好处是:
34      * 如果具体子类没有某个创建方法, 则会自动调用父类的空方法,
35      * 而抽象方法则强制子类实现, 造成编程上的冗余
36      */
37     public void createMainProduct() {
38         // 空方法
39     }
40
41     public void createAdd1Product() {
42         // 空方法
43     }
44
45     public void createAdd2Product() {
46         // 空方法
47     }
48
49 }
```

注意代码中的三个`create`方法并没有定义为抽象的, 而是定义成了空方法, 这样其子类就可以不必将三个方法全部实现, 而只需要重写其感兴趣的方法即可, 请参照注释仔细体会其用意。

TV建造者实现类:

代码片段11 GoodsBuilderImplTV.java

```
1 package cn.steven.pattern.demo.builder.quest;
2
3 /**
4  * 电视机创建者
5  */
6 public class GoodsBuilderImplTV extends GoodsBuilder {
7
8     /**
9      * 创建主要商品
10     */
11     public void createMainProduct() {
12         getProductList().add(new Product("TV", 2999.99f));
13     }
14
15     /**
16      * 创建标准版的附加商品
17     */
18     public void createAdd1Product() {
19         getProductList().add(new Product("演示DVD", 5.99f));
20     }
21
22     /**
23      * 创建豪华版的附加商品
24     */
25 }
```

```

25     public void createAdd2Product() {
26         getProductList().add(new Product("蓝牙耳机", 199.05f));
27     }
28 }

```

PSP建造者实现类:

代码片段12 GoodsBuilderImplPSP.java

```

1  package cn.steven.pattern.demo.builder.quest;
2
3  /**
4   * PSP创建者
5   */
6  public class GoodsBuilderImplPSP extends GoodsBuilder {
7
8      /**
9       * 创建主要商品
10      */
11     public void createMainProduct() {
12         getProductList().add(new Product("PSP", 1499.99f));
13     }
14
15     /**
16      * 创建标准版的附加商品
17      */
18     public void createAdd1Product() {
19         getProductList().add(new Product("贴膜", 1.0f));
20         getProductList().add(new Product("充电器", 99f));
21     }
22
23     /**
24      * 创建豪华版的附加商品
25      */
26     public void createAdd2Product() {
27         getProductList().add(new Product("耳机", 30.99f));
28         getProductList().add(new Product("硅胶套", 25.9f));
29     }
30 }

```

注意代码片段12中第19行~第20行与第27行~第28行，它们说明每一个创建方法创建的对象个数可以是任意的。

抽象导演类:

代码片段13 GoodsDirector.java

```

1  package cn.steven.pattern.demo.builder.quest;
2
3  import java.util.List;
4
5  /**
6   * 抽象导演类
7   */

```



```
8 abstract public class GoodsDirector {
9
10     /**
11      * 聚合创建对象
12      */
13     private GoodsBuilder builder;
14
15     public GoodsBuilder getBuilder() {
16         return builder;
17     }
18
19     public void setBuilder(GoodsBuilder builder) {
20         this.builder = builder;
21     }
22
23     /**
24      * 构造方法
25      *
26      * @param builder
27      */
28     public GoodsDirector(GoodsBuilder builder) {
29         super();
30         this.builder = builder;
31     }
32
33     /**
34      * 构建产品
35      */
36     public abstract List<Product> construct();
37 }
```

标准版产品导演类:

代码片段14 GoodsDirectorImplCommon.java

```
1 package cn.steven.pattern.demo.builder.quest;
2
3 import java.util.List;
4
5 /**
6  * 标准版导演类
7  */
8 public class GoodsDirectorImplCommon extends GoodsDirector {
9
10     /**
11      * 构造方法
12      *
13      * @param builder
14      */
15     public GoodsDirectorImplCommon(GoodsBuilder builder) {
16         super(builder);
17     }
18 }
```

```

19  /**
20   * 重写父类的构建方法
21   */
22  @Override
23  public List<Product> construct() {
24      //
25      /**
26       * 初始化创建对象
27       */
28      getBuilder().init();
29      //
30      /**
31       * 创建过程
32       */
33      getBuilder().createMainProduct();
34      getBuilder().createAdd1Product();
35      //
36      /**
37       * 得到产品
38       */
39      return getBuilder().getProductList();
40      //
41  }
42
43  }

```

豪华版产品导演类:

代码片段15 GoodsDirectorImplCostly.java

```

1  package cn.steven.pattern.demo.builder.quest;
2
3  import java.util.List;
4
5  /**
6   * 豪华版导演类
7   */
8  public class GoodsDirectorImplCostly extends GoodsDirector {
9
10     /**
11      * 构造方法
12      *
13      * @param builder
14      */
15     public GoodsDirectorImplCostly(GoodsBuilder builder) {
16         super(builder);
17     }
18
19     /**
20      * 重写父类的构建方法
21      */
22     @Override

```

```

23     public List<Product> construct() {
24
25         /**
26          * 初始化创建对象
27          */
28         getBuilder().init();
29
30         /**
31          * 创建过程
32          */
33         getBuilder().createMainProduct();
34         getBuilder().createAdd1Product();
35         getBuilder().createAdd2Product();
36
37         /**
38          * 得到产品
39          */
40         return getBuilder().getProductList();
41     }
42 }
43
44 }

```

由以上两个导演实现类可以看出，产品的创建者没有增加，但是通过不同的导演类却可以生产出不同的产品组合，这就极大地增加了设计组合的灵活性。

客户端代码：

代码片段16 Client.java

```

1  package cn.steven.pattern.demo.builder.quest;
2
3  import java.util.List;
4
5  /**
6   * 客户端代码
7   */
8  public class Client {
9
10     public static void main(String[] args) {
11         /**
12          * 导演对象
13          */
14         GoodsDirector director;
15
16         /**
17          * 标准版TV
18          */
19         director = new GoodsDirectorImplCommon(
20             new GoodsBuilderImplTV());
21         System.out.println("标准版TV:");
22         Client.viewList(director.construct());
23     }

```



```

24  /**
25   * 标准版PSP
26   */
27  director = new GoodsDirectorImplCommon(
28      new GoodsBuilderImplPSP());
29  System.out.println("标准版PSP:");
30  Client.viewList(director.construct());
31
32  /**
33   * 豪华版TV
34   */
35  director = new GoodsDirectorImplCostly(
36      new GoodsBuilderImplTV());
37  System.out.println("豪华版TV:");
38  Client.viewList(director.construct());
39
40  /**
41   * 豪华版PSP
42   */
43  director = new GoodsDirectorImplCostly(
44      new GoodsBuilderImplPSP());
45  System.out.println("豪华版PSP:");
46  Client.viewList(director.construct());
47
48  }
49
50  /**
51   * 打印列表信息的工具方法
52   *
53   * @param list
54   */
55  public static void viewList(List<Product> list) {
56      for (Product product : list) {
57          System.out.print(product.toString());
58      }
59      System.out.println();
60  }
61  }

```

运行结果如图6-7所示。



图6-7 Client.java运行结果

对照代码片段16和图6-7可见，以上设计代码已经完全可以实现各种商品的捆绑销售，如果要增加产品，则只要再编写一个Builder即可，如果要增加一个简易版导演，也可以简单地编写一个Director的子类来解决。

6.2.3 建造者模式的实际使用案例

建造者模式的使用在JDK中并不明显，不过在实际的编程中却经常见到。以下即为真实的商业案例。

案例一：简易加密。比如有一个明文需要加密后传送出去，假设明文为“go now”，现在有很多种加密方式可以选择：第一种是采用明文字母字典顺序的下一个字母来替代明文做密文，上述明文就变为“hp opx”，第二种是采用明文字母字典顺序的前一个字母来替代明文做密文，上述明文就变为“fn mnv”，类似的有很多种加密方法，则实现这种逻辑就可以使用建造者模式。

案例二：多种网络接入方案。在现在的网站技术中，接入的渠道越来越多了，技术也越来越先进，如WAP、SMS、EMAIL、传统的Web、Socket等。怎样才能保证在添加新的网络接入渠道时不需要更多地修改代码甚至不改代码呢？使用建造者模式就可以很好地解决这个问题，这是因为不同的接入方案所使用的操作步骤是相同的，可使用Director来控制步骤；而不同的接入方案操作的具体实现是不同的，将其封装到不同的Builder中，就可以解决此问题。

案例三：数据的多种展示方式。如有大量的数据要展示，但是需要多种不同的展示效果，比如有的效果要求数字保留两位小数，有的又不需要；有的需要图片居中显示，有的要求图片加边框等，也就是说不同的显示效果对于某个具体的数据的处理是不同的，但是它们都使用相同的顺序来处理数据，这种情况下使用建造者模式是十分合适的。

6.2.4 建造者模式的使用范围

建造者模式的实现要点如下：

- 建造者模式主要用于“分步骤构建一个复杂的对象”。在这其中，“分步骤”是一个稳定的算法，而复杂对象的各个部分则经常变化。
- 产品不需要抽象类，特别是由于创建对象的算法复杂而导致使用此模式的情况下或者此模式应用于产品的生成过程，其最终结果可能差异很大，不大可能提炼出一个抽象产品类。
- 创建者中的创建子部件的接口方法不是抽象方法而是空方法，不进行任何操作，具体的创建者只需要覆盖需要的方法就可以，但是这也不是绝对的，特别是类似文本转换这种情况下，默认的方法将输入原封不动的输出是合理的默认操作。

• 前面我们说过的抽象工厂模式（Abstract Factory）解决“系列对象”的需求变化，Builder模式解决“对象部分”的需求变化，建造者模式常和组合模式（Composite Pattern）结合使用。

建造者模式的使用效果如下：

- 使用建造者模式使得产品的内部表象可以独立变化，可以使客户端不必知道产品内部组成的细节。
- 每一个Builder都相对独立，与其他的Builder无关。
- 可以更加精细地控制构造过程。
- 将构建代码和表示代码分开。

建造者模式的适用性如下:

- 需要生成的产品对象有复杂的内部结构。
- 需要生成的产品对象的属性相互依赖, 建造者模式可以强制制定生成顺序。
- 在对象创建过程中会使用到系统中的一些其他对象, 这些对象在产品对象的创建过程中不易得到。

6.2.5 与其他模式的关系

建造者模式与工厂模式的区别:

- 简单工厂模式: 又叫静态工厂方法模式, 它定义一个具体的工厂类, 通过用静态方法来负责创建一些类的实例。也就是说, 这个类集合了部分功能的类似或近似类的实例化, 但工厂类无法被继承 (只有一个工厂类, 通过该类中静态方法来创建产品类的对象, 随着产品类的增多, 该静态方法也越来越复杂和难以维护)。
- 工厂方法模式: 通过一个工厂类来完成对象的实例化。工厂模式在调用的时候需要先实例化工厂类, 再通过工厂类来返回一个子公司对象类, 工序发生了变化。根据不同的产品创建不同的工厂类来返回实例。
- 抽象工厂模式: 它分为抽象工厂类 (实现抽象工厂类的具体返回实例的工厂类)、抽象产品类、具体产品类。和工厂模式差不多, 区别就是抽象工厂模式把一系列的产品进行统一。抽象工厂模式中, 我们会把一系列相似的产品放在一个工厂类里面实例化, 和现在车间的流水线差不多, 一个流水线负责生产相似的产品, 不同的产品需要另一条流水线来生产。工厂模式是针对细微的产品来创建工厂类。

- 建造者模式: 强调的是控制产品的生产过程, 使不同的产品生产可以使用相同的步骤。

在实际应用中, 建造者模式常与不同的工厂模式配合使用。

6.3 建造者模式总结

建造者模式主要用于创建一些复杂的对象, 这些对象内部的建造顺序通常是稳定的, 但对象内部的构建通常面临复杂的变化。建造者模式的好处就是使得构造代码与表示代码分离, 由于建造者隐藏了该产品是如何组装的, 所以若需要改变一个产品的内部表示, 只需要再定义一个具体的建造者就可以了。

有些情况下, 一个对象会有一些重要的性质, 在它们没有恰当的值之前, 对象不能作为一个完整的产品使用。比如, 一个电子邮件有发件人地址、收件人地址、主题、内容、附录等部分, 而在最起码的收件人地址未被赋值之前, 这个电子邮件不能发出。

有些情况下, 一个对象的一些性质必须按照某个顺序赋值才有意义。在某个性质没有赋值之前, 另一个性质则无法赋值。这些情况使得性质本身的建造涉及复杂的商业逻辑。

这时候, 此对象相当于一个有待建造的产品, 而对象的这些性质相当于产品的零件, 建造产品的过程就是组合零件的过程。由于组合零件的过程很复杂, 因此, 这些“零件”的组合过程往往被“外部化”到一个称做建造者的对象里, 建造者返还给客户端的是一个全部零件都建造完毕的产品对象。

当需要控制创建对象的过程时, 需要使用建造者模式。

第7章 原型模式 (Prototype)

原型模式使用用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象¹。在生活中有时生成对象的方法是“照葫芦画瓢”，它的意思就是使用一个现成的物体来进行仿制，如果用于生物上也就是“克隆”。

克隆是英文clone的音译，简单讲就是一种人工诱导的无性繁殖方式。但克隆与无性繁殖是不同的。无性繁殖是指不经过雌雄两性生殖细胞的结合，只由一个生物体产生后代的生殖方式，常见的有孢子生殖、出芽生殖和分裂生殖。由植物的根、茎、叶等经过压条或嫁接等方式产生新个体也叫无性繁殖。绵羊、猴子和牛等动物没有人工操作是不能进行无性繁殖的。科学家把通过人工遗传操作动物繁殖的过程叫克隆，这门生物技术叫克隆技术。

简单来说，就是利用复制的技术来实现对象的复制。在前面章节的学习中，可以发现整个创建型模式都是解决对象创建问题的，但是其他的几个模式都是通过新创建一个对象或者共享对象的方式得到对象。在某些情况下如果一个对象本身的数据十分复杂，在构建出来之后又进行了大量的属性设置和修改的动作，那么如果要对这个对象进行复制的话，使用通过类新建对象的方式就不合适了，显然强制要求复制的话，共享也不合适，这个时候就需要使用原型模式来解决对象的克隆问题了。

7.1 新连锁店开张

连锁机构是现在非常流行的商业模式，比如耳熟能详的肯德基、麦当劳、沃尔玛等，很多行业都可以应用这种模式来发展规模，如图7-1所示就是连锁经营的一个基本思路。

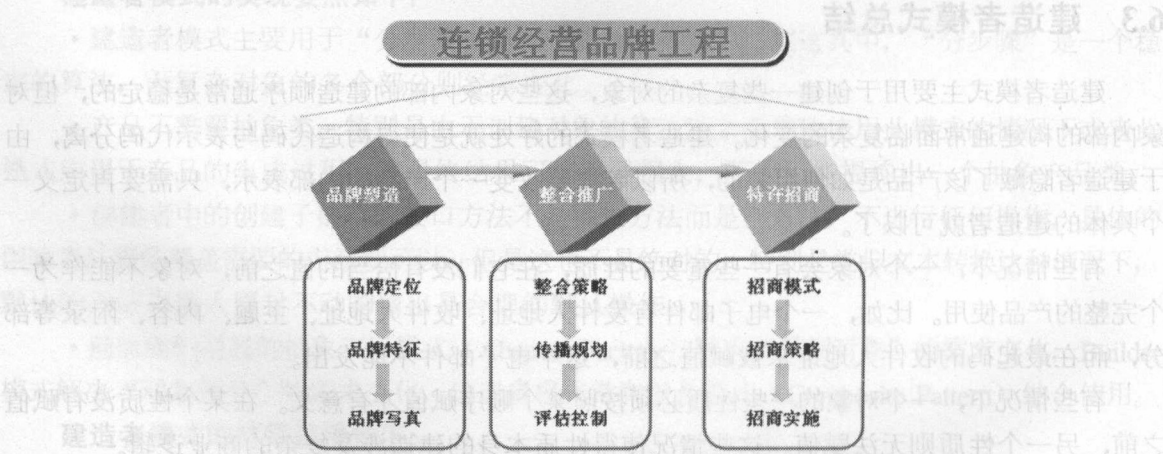


图7-1 连锁经营思路

随着超市的发展，新的连锁店铺要开张了，如何创建一个新的店铺呢？

¹Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. OF[95]

新的店铺自然是不能从零做起，因为这样太耗时间了，一个普通的做法是设计一个店铺类，每开一个连锁店就从这个类中实例化一个对象。

店铺所用的商品代码如下：

代码片段1 Goods.java

```
1 package cn.steven.pattern.demo.prototype;
2
3 /**
4  * 商品
5  */
6 public class Goods {
7     /** shop1.setGoodsList(glist);
8     * 商品名
9     */
10    private String name;
11
12    /** Shop shop2 = new Shop();
13    * 商品价格 Name("分店1");
14    */
15    private double price;
16
17    public String getName() {
18        return name;
19    }
20
21    public void setName(String name) {
22        this.name = name;
23    }
24
25    public double getPrice() {
26        return price;
27    }
28
29    public void setPrice(double price) {
30        this.price = price;
31    }
32
33    /**
34     * 构造方法
35     * @param name
36     * @param price
37     */
38    public Goods(String name, double price) {
39        super();
40        this.name = name;
41        this.price = price;
42    }
43
44    // ...
45 }
```

店铺代码如下:

代码片段2 Shop.java

```
1 package cn.steven.pattern.demo.prototype;
2
3 import java.util.List;
4
5 /**
6  * 店铺
7  */
8 public class Shop {
9
10     /**
11      * 店铺名
12      */
13     private String name;
14
15     /**
16      * 商品列表
17      */
18     private List<Goods> goodsList;
19
20     public String getName() {
21         return name;
22     }
23
24     public void setName(String name) {
25         this.name = name;
26     }
27
28     public List<Goods> getGoodsList() {
29         return goodsList;
30     }
31
32     public void setGoodsList(List<Goods> goodsList) {
33         this.goodsList = goodsList;
34     }
35
36 }
```

注意在店铺代码中聚合了商品的集合。
使用原始方法创建连锁店的方法如下:

代码片段3 Step_1.java

```
1 package cn.steven.pattern.demo.prototype;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 实例化类的方法
```



```

8  */
9  public class Step_1 {
10
11      public static void main(String[] args) {
12
13          /**
14           * 现有的一个总店对象
15           */
16          Shop shop1 = new Shop();
17          shop1.setName("总店");
18          List<Goods> glist = new ArrayList<Goods>();
19          glist.add(new Goods("面包", 5.5));
20          glist.add(new Goods("香肠", 6.9));
21          shop1.setGoodsList(glist);
22
23          /**
24           * 连锁店
25           */
26          Shop shop2 = new Shop();
27          shop2.setName("分店1");
28          List<Goods> glist2 = new ArrayList<Goods>();
29          glist2.add(new Goods("面包", 5.5));
30          glist2.add(new Goods("香肠", 6.9));
31          shop2.setGoodsList(glist2);
32      }
33  }

```

由代码片段3中第16行~第31行可以看出, 虽然这些代码可以生成一个新的商店对象, 但是代码的重用性非常低, 其原因就是此对象生成的时候没有设置属性, 都需要由外界去进行设置, 所以导致一个新对象生成之后有大量的工作要做。

对于这种不适合新构造对象却又需要复制新对象的情况就适合使用原型模式了。

7.2 原型模式的结构

原型模式使用了克隆的方式产生新的对象, 新生成的对象也可以根据需求稍做修改以适应需求。

7.2.1 原型模式

原型模式允许一个对象再创建另外一个可定制的对象, 根本无需知道任何创建的细节。其工作原理是: 将一个原型对象传给那个要发动创建的对象, 这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。

实际运用过程中通常通过在类中实现一个可以复制的clone方法来进行原型模式的设计, 其类图如图7-2所示。

原型模式的类图比较简单, 其参与者如下所示。

- **Prototype**接口: 声明一个克隆自身的接口。
- **SampleClass**、**ComplexClass**: 实现了克隆方法的类。

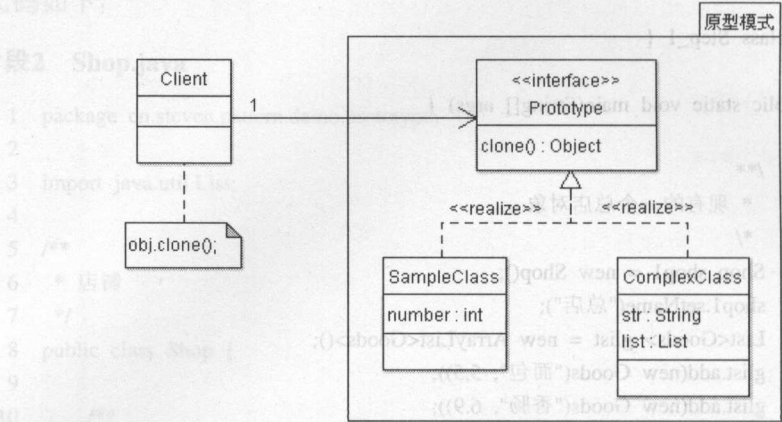


图7-2 原型模式类图

• **Client:** 让一个原型克隆自身从而创建一个新的对象。

其具体代码参考如下。

原型接口:

代码片段4 Prototype.java

```
1 package cn.steven.pattern.demo.prototype;
2
3 /**
4  * 原型模式接口
5  */
6 public interface Prototype {
7
8     /**
9      * 具体克隆方法
10     * @return 新生成的对象
11     */
12     Prototype clone();
13 }
```

具体的实现类如下。

简易属性类:

代码片段5 SampleClass.java

```
1 package cn.steven.pattern.demo.prototype;
2
3 /**
4  * 实现了原型模式的简单类
5  */
6 public class SampleClass implements Prototype {
7
8     /**
9      * 一个简单属性
10     */
11     private int i;
12 }
```

```

13     public int getI() {
14         return i;
15     }
16
17     public void setI(int i) {
18         this.i = i;
19     }
20
21     /**
22      * 构造方法
23      */
24     public SampleClass(int i) {
25         super();
26         this.i = i;
27     }
28
29     @Override
30     public Prototype clone() {
31         return new SampleClass(this.getI());
32     }
33
34     @Override
35     public String toString() {
36         return Integer.toString(this.getI());
37     }
38
39 }

```

复杂属性类:

代码片段6 ComplexClass.java

```

1  package cn.steven.pattern.demo.prototype;
2
3  import java.util.List;
4
5  /**
6   * 实现了原型模式的复杂类
7   */
8  public class ComplexClass implements Prototype {
9
10     /**
11      * 一个字符串
12      */
13     private String str;
14
15     /**
16      * 一个集合
17      */
18     private List list;
19
20     public String getStr() {

```



```

21         return str;
22     }
23
24     public void setStr(String str) {
25         this.str = str;
26     }
27
28     public List getList() {
29         return list;
30     }
31
32     public void setList(List list) {
33         this.list = list;
34     }
35
36     @Override
37     public Prototype clone() {
38         ComplexClass cc = new ComplexClass();
39         cc.setList(this.getList());
40         cc.setStr(this.getStr());
41         return cc;
42     }
43
44     @Override
45     public String toString() {
46         return this.getStr() + " " + this.getList();
47     }
48
49 }

```

注意上述代码中用于比较对象的方法，这个知识点是Java学习过程中一个很重要的问题。

==和equals

- 对于基本类型（boolean, char, byte, short, int long float, double），只能用==。
- 对于基本类型的封装类（Boolean, Integer, Double, Long, Float），==比较的是地址，equals比较的是值。

• 如果是类对象，那么在对象的类没有覆盖equals()方法的时候，它们是相同的，就是比较两方的地址（如果覆盖了equals，那么根据实际覆盖的内容比较）。

- String比较特殊，==比较地址，equals比较值。

注意，在重写equals方法时，要满足离散数学上的特性：

- 自反性：对任意引用值X，x.equals(x)的返回值一定为true。
- 对称性：对于任何引用值x，y，当且仅当y.equals(x)返回值为true时，x.equals(y)的返回值一定为true。
- 传递性：如果x.equals(y)=true，y.equals(z)=true，则x.equals(z)=true。
- 一致性：如果参与比较的对象没有任何改变，则对象比较的结果也不应该有任何改变。
- 非空性：任何非空的引用值X，x.equals(null)的返回值一定为false。

在学习原型模式时，需要清楚新建的对象和对象中的属性和源对象的关系。

客户端代码：

代码片段7 Client_1.java

```
1 package cn.steven.pattern.demo.prototype;
2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 /**
7  * 客户代码
8  */
9 public class Client_1 {
10     public static void main(String[] args) {
11
12         /**
13          * 克隆简单对象
14          */
15         SampleClass sc = new SampleClass(99);
16         Prototype scc = sc.clone();
17
18         /**
19          * 判断对象是否是相同的指针
20          */
21         System.out.println("sc==scc ? " + (sc == scc));
22
23         /**
24          * 判断值是否相等
25          */
26         System.out.println("sc: " + sc.toString());
27         System.out.println("scc: " + scc.toString());
28
29         /**
30          * 克隆复杂对象
31          */
32         ComplexClass cc = new ComplexClass();
33         cc.setStr("一个字符串");
34         List list = new ArrayList();
35         list.add("字符串");
36         list.add(25);
37         cc.setList(list);
38
39         Prototype ccc = cc.clone();
40
41         /**
42          * 判断对象是否是相同的指针
43          */
44         System.out.println("cc==ccc ? " + (cc == ccc));
45
46         /**
47          * 判断值是否相等
48          */
```

```

47 System.out.println("cc: " + cc.toString());
48 System.out.println("ccc: " + ccc.toString());
49 }
50 }

```

运行结果如图7-3所示。

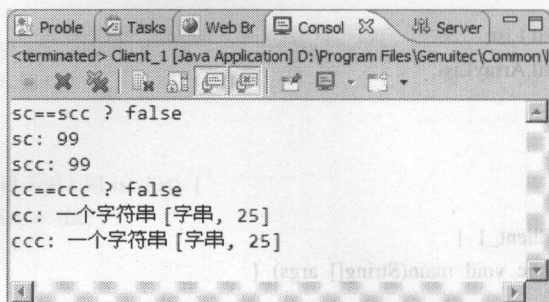


图7-3 运行结果

由结果可见，生成后的对象和源对象在内存中的确是两个不同的地址，而且生成后的对象与源对象中的属性是完全一样的。

现在读者可能会问一个问题，就是虽然新对象和源对象不是一样的对象，但是其中的属性值和源对象的值有可能是共享一个对象，也有可能是复制的值，这个问题将在下一节“浅克隆与深克隆”中讲解。

7.2.2 浅克隆与深克隆

浅克隆与深克隆的区别：

- 浅克隆：被复制的所有变量都具有与原来对象相同的值，而所有的对其他对象的引用都仍然指向原来的对象。
- 深克隆：把引用对象的变量指向复制过的新对象，而不是原有的被引用的对象。

Java是纯面向对象的程序设计语言。Java里，所有的类的顶级父类都是java.lang.Object类，也就是说，如果一个类没有显示申明继承关系，它的父类默认就是java.lang.Object。

在类Object中定义了clone方法，首先来看一下此方法的api文档说明：

```

protected Object clone()
throws CloneNotSupportedException

```

创建并返回此对象的一个副本。“副本”的准确含义可能依赖于对象的类。这样做的目的是，对于任何对象 x，表达式：

```
x.clone() != x
```

为true，表达式：

```
x.clone().getClass() == x.getClass()
```

也为true，但这些并非必须要满足的要求。一般情况下：

```
x.clone().equals(x)
```

为true，但这并非必须要满足的要求。

按照惯例, 返回的对象应该通过调用`super.clone`获得。如果一个类及其所有的超类 (Object 除外) 都遵守此约定, 则`x.clone().getClass() == x.getClass()`。

按照惯例, 此方法返回的对象应该独立于该对象 (正被复制的对象)。要获得此独立性, 在`super.clone`返回对象之前, 有必要对该对象的一个或多个字段进行修改。这通常意味着要复制包含正在被复制对象的内部“深层结构”的所有可变对象, 并使用对副本的引用替换对这些对象的引用。如果一个类只包含基本字段或对不变对象的引用, 那么通常不需要修改`super.clone`返回的对象中的字段。

Object类的`clone`方法执行特定的复制操作。首先, 如果此对象的类不能实现接口`Cloneable`, 则会抛出`CloneNotSupportedException`。注意, 所有的数组都被视为可实现接口`Cloneable`。否则, 此方法会创建此对象的类的一个新实例, 并像通过分配那样, 严格使用此对象相应字段的内容初始化该对象的所有字段; 这些字段的内容没有被自我复制。所以, 此方法执行的是该对象的“浅表复制”, 而不是“深层复制”操作。

Object类本身不实现接口`Cloneable`, 所以在类为Object的对象上调用`clone`方法将会导致在运行时抛出异常。

返回:

此实例的一个副本。

抛出:

`CloneNotSupportedException`——如果对象的类不支持`Cloneable`接口, 则重写`clone`方法的子类也会抛出此异常, 以指示无法复制某个实例。

常用的浅克隆的例子如下:

代码片段8 TestClone.java

```
1 package cn.steven.pattern.demo.prototype.pattern;
2
3 /**
4  * 测试浅克隆
5  */
6 public class TestClone {
7     public static void main(String[] args) {
8         MyClone myClone1 = new MyClone("clone1");
9
10        MyClone myClone2 = (MyClone) myClone1.clone();
11
12        if (myClone2 != null) {
13            System.out.println(myClone2.getName());
14            System.out.println("Clone2 equals Clone1: "
15                + myClone2.equals(myClone1));
16        } else {
17            System.out.println("不支持克隆");
18        }
19    }
20
21 }
22
```

```

23 class MyClone {
24     private String name;
25
26     public MyClone(String name) {
27         this.name = name;
28     }
29
30     public String getName() {
31         return name;
32     }
33
34     public void setName(String name) {
35         this.name = name;
36     }
37
38     /**
39      * 实现的克隆方法
40      */
41     public Object clone() {
42         try {
43             /**
44              * 常用的克隆调用方法
45              */
46             return super.clone();
47         } catch (CloneNotSupportedException e) {
48             return null;
49         }
50     }
51 }

```

运行结果如图7-4所示。

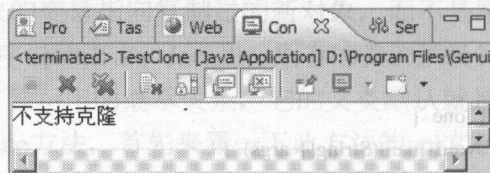


图7-4 错误的浅克隆

说明MyClone#clone()方法调用super.clone()时抛出了CloneNotSupportedException异常，不支持克隆。

为什么父类java.lang.Object里提供了clone()方法，却不能调用呢？

原来，Java语言虽然提供了这个方法，但考虑到安全问题，一方面将clone()方法的访问级别设置为protected型，以限制外部类访问；另一方面，强制需要提供clone功能的子类实现java.lang.Cloneable接口。在运行期，JVM会检查调用clone()方法的类，如果该类未实现java.lang.Cloneable接口。则抛出CloneNotSupportedException异常。

java.lang.Cloneable接口是一个空的接口，没有申明任何属性与方法。该接口只是告诉JVM，该接口的实现类需要开放“克隆”功能。

我们再将MyClone类稍做改变，让其实现Cloneable接口：

代码片段9 修改方案

```
1 ...
2 class MyClone implements Cloneable{
3 ...
4 }
```

下面再次运行后的结果如图7-5所示。

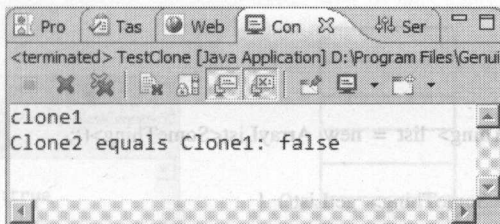


图7-5 正确结果

根据结果，我们可以发现：

- myClone1.clone()克隆了跟myClone1具有相同属性值的对象。
- 克隆出的对象myClone2跟myClone1不是同一个对象（具有不同的内存空间）。

注意：如果一个类需要可以被克隆，需要实现clone方法和实现Cloneable接口，二者缺一不可。

上例说明了怎么样克隆一个具有简单属性（String，int，boolean等）的对象。但如果一个对象的属性类型是List，Map，或者用户自定义的其他类时，克隆行为是通过怎样的方式进行的呢？

很多时候，我们希望即使修改了克隆后的对象的属性值，也不会影响到源对象，这种克隆我们称之为对象的深克隆。怎么样实现对象的深克隆呢？

代码片段10 TestClone2.java

```
1 package cn.steven.pattern.demo.prototype.pattern;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 测试深克隆
8  */
9 public class TestClone2 {
10     public static void main(String[] args) {
11
12         // 源对象
13         DeepClone dc = new DeepClone();
14         dc.getList().add(new Something("一个对象"));
15
16         // 克隆对象
17         DeepClone dcc = (DeepClone) dc.clone();
```



```

18 class MyClone {
19     // 判断对象是否为同一对象
20     System.out.println("DeepClone是否为同一对象:" + (dc == dcc));
21     System.out.println("DeepClone::list是否为同一对象:"
22         + (dc.getList() == dcc.getList()));
23 }
24 }
25
26 class DeepClone implements Cloneable {
27
28     /**
29      * 一个集合
30      */
31     List<Something> list = new ArrayList<Something>();
32
33     public List<Something> getList() {
34         return list;
35     }
36
37     public void setList(List<Something> list) {
38         this.list = list;
39     }
40
41     /**
42      * 实现的克隆方法
43      */
44     public Object clone() {
45         try {
46             /**
47              * 常用的克隆调用方法
48              */
49             return super.clone();
50         } catch (CloneNotSupportedException e) {
51             return null;
52         }
53     }
54 }
55
56 class Something {
57     private String name;
58
59     public String getName() {
60         return name;
61     }
62
63     public void setName(String name) {
64         this.name = name;
65     }
66
67     public Something(String name) {
68         super();

```

```
69         this.name = name;
70     }
71 }
```

注意代码片段10中的第31行表示在被克隆的对象内部拥有一个集合，此集合中的对象为自定义的类对象。其运行结果如图7-6所示。

为了便于理解，可以参考以下内存示意图，如图7-7所示。

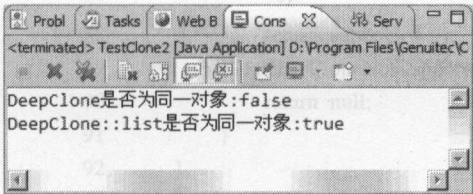


图7-6 深克隆结果1

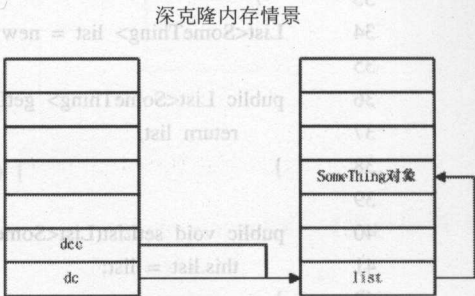


图7-7 深克隆内存情景1

由图7-7可见，上述代码实现的深克隆虽然实现了对象的克隆，但是对象中的集合对象却是共享形态的，这样会导致dc的list改变了，dcc的list也将被改变，这就不是深克隆的意图了，其意图是将一个对象完全复制为另一个对象。

根据深克隆的要求，代码需修改为如下：

代码片段11 TestClone2.java修改版

```
1 package cn.steven.pattern.demo.prototype.pattern;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 测试深克隆
8  */
9 public class TestClone2 {
10     public static void main(String[] args) {
11
12         // 源对象
13         DeepClone dc = new DeepClone();
14         dc.getList().add(new Something("一个对象"));
15
16         // 克隆对象
17         DeepClone dcc = (DeepClone) dc.clone();
18
19         // 判断对象是否为同一对象
20         System.out.println("DeepClone是否为同一对象:" + (dc == dcc));
21         System.out.println("DeepClone::list是否为同一对象:" +
22             + (dc.getList() == dcc.getList()));
23
24         System.out.println("DeepClone::list[0]是否为同一对象:" +
```

```

25         (dc.getList().get(0) == dcc.getList().get(0));
26     }
27 }
28
29 class DeepClone implements Cloneable {
30
31     /**
32      * 一个集合
33      */
34     List<Something> list = new ArrayList<Something>();
35
36     public List<Something> getList() {
37         return list;
38     }
39
40     public void setList(List<Something> list) {
41         this.list = list;
42     }
43
44     /**
45      * 实现的克隆方法
46      */
47     public Object clone() {
48         try {
49             // 常用的克隆调用方法
50             // 实现的方法
51             public Object clone() {
52                 DeepClone dc = (DeepClone)super.clone();
53                 dc.setList(new ArrayList<Something>());
54                 for (Something something : list) {
55                     dc.getList().add((Something)something.clone());
56                 }
57                 return dc;
58             } catch (CloneNotSupportedException e) {
59                 return null;
60             }
61         }
62     }
63
64     class Something implements Cloneable {
65         private String name;
66         public String getName() {
67             return name;
68         }
69
70         public void setName(String name) {
71             this.name = name;
72         }
73     }
74
75     public Something(String name) {

```



```
76     super();
77     this.name = name;
78 }
79
80 /**
81  * 实现的克隆方法
82  */
83 public Object clone() {
84     try {
85         /**
86          * 常用的克隆调用方法
87          */
88         return super.clone();
89     } catch (CloneNotSupportedException e) {
90         return null;
91     }
92 }
93 }
```

代码片段11中修改的地方为：第47行~第61行修改了clone方法，第83行~第93行增加了clone方法。
其运行结果如图7-8所示。其内存情景如图7-9所示。

深克隆内存情景

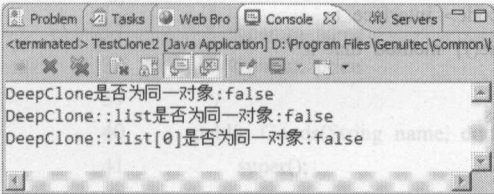


图7-8 深克隆结果2

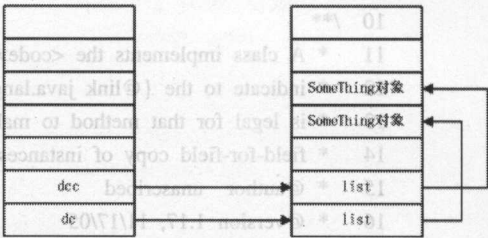


图7-9 深克隆内存情景2

由图7-8可见，代码经过修改现在已经可以完全满足深克隆的要求了。在实际的企业项目中，使用深克隆的需求比较多，请仔细体会浅克隆和深克隆的区别，加以熟练应用。

7.2.3 使用原型模式解决连锁店问题

经过上一节的学习，对于新连锁店的建设问题，经分析发现连锁店所有的相关实体都不可以共享使用，所以需要使用深克隆技术。可以用如下方法设计类图，如图7-10所示。

图7-10中要注意到Shop和Goods都实现了克隆方法，但是类图中还有一个默认的父类Object并没有画出，读者在思考的时候要要和图7-2加以对比理解。

Cloneable是JDK提供的声明式接口，没有定义方法。

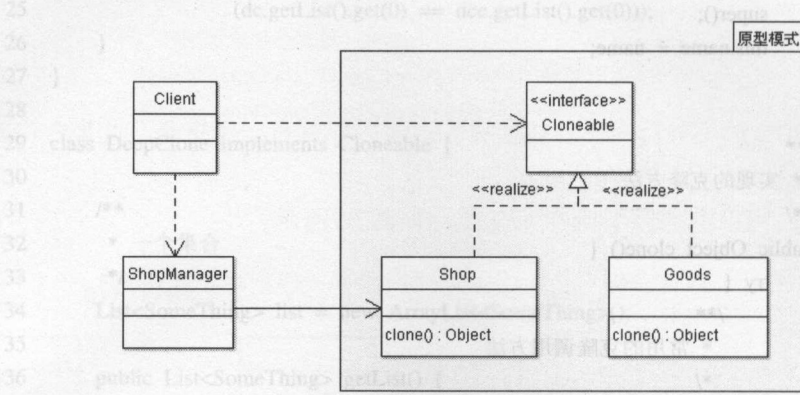


图7-10 使用原型模式的类图

代码片段12 Cloneable.java

```

1  /*
2   * @(#)Cloneable.java      1.17 05/11/17
3   *
4   * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
5   * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
6   */
7
8  package java.lang;
9
10 /**
11  * A class implements the <code>Cloneable</code> interface to
12  * indicate to the {@link java.lang.Object#clone()} method that it
13  * is legal for that method to make a
14  * field-for-field copy of instances of that class.
15  * @author  unascribed
16  * @version 1.17, 11/17/05
17  * @see    java.lang.CloneNotSupportedException
18  * @see    java.lang.Object#clone()
19  * @since   JDK1.0
20  */
21 public interface Cloneable {
22 }

```

具备克隆功能的商品类:

代码片段13 Goods.java

```

1  package cn.steven.pattern.demo.prototype.quest;
2
3  /**
4   * 商品，具备克隆功能
5   */
6  public class Goods implements Cloneable {
7
8      /**
9       * 商品名

```

```

10      */
11      private String name;
12      * 实现克隆方法
13      /**
14      * 商品价格
15      */
16      private double price;
17      * 克隆调用方法
18      public String getName() {
19          return name;
20      }
21      * 克隆方法
22      public void setName(String name) {
23          this.name = name;
24      }
25      * 克隆方法
26      public double getPrice() {
27          return price;
28      }
29
30      public void setPrice(double price) {
31          this.price = price;
32      }
33      * 克隆方法
34      /**
35      * 构造方法
36      *
37      * @param name
38      * @param price
39      */
40      public Goods(String name, double price) {
41          super();
42          this.name = name;
43          this.price = price;
44      }
45
46      /**
47      * 实现的克隆方法
48      */
49      public Object clone() {
50          try {
51              /**
52              * 常用的克隆调用方法
53              */
54              return super.clone();
55          } catch (CloneNotSupportedException e) {
56              return null;
57          }
58      }
59      * 用于保存所有的店铺，是个查询和克隆
60      }

```


代码片段13展示类由于只具备简单属性，所以使用浅克隆即可。
具备深克隆功能的Shop类：

代码片段14 Shop.java

```
1 package cn.steven.pattern.demo.prototype.quest;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 /**
7  * 店铺，具备克隆功能
8  */
9 public class Shop implements Cloneable {
10
11     /**
12      * 店铺名，作为ID
13      */
14     private String name;
15
16     /**
17      * 商品列表
18      */
19     private List<Goods> goodsList = new ArrayList<Goods>();
20
21     public String getName() {
22         return name;
23     }
24
25     public void setName(String name) {
26         this.name = name;
27     }
28
29     public List<Goods> getGoodsList() {
30         return goodsList;
31     }
32
33     public void setGoodsList(List<Goods> goodsList) {
34         this.goodsList = goodsList;
35     }
36
37     /**
38      * 因为每一个店铺都有一个唯一的名称，
39      * 所以有必要用一个方法作为其克隆之后的设置方法
40      *
41      * @param name
42      *      唯一的店铺名称
43      */
44     public void init(String name) {
45         this.setName(name);
46     }
```

```

47
48 /** // 生成总店 = qmMap>shopMap>
49 * 实现深克隆方法 new Shop();
50 */ shop.setName("DPC连锁总店");
51 public Object clone() { add(new Goods("鱼干薯", 299.99));
52 try { getGoodsList().add(new Goods("葡萄酒", 500.5));
53 /** GoodsList().add(new Goods("挂面", 1.5));
54 * 深克隆调用方法
55 // 抛*/
56 Shop shop = (Shop) super.clone();
57 shop.setGoodsList(new ArrayList<Goods>());
58 for (Goods g : goodsList) {
59 ShopManager shop.getGoodsList().add((Goods) g.clone());
60 }
61 // return shop;
62 } catch (CloneNotSupportedException e) {
63 return null;
64 } myShop.init("DPC连锁总店");
65 } myShop.getGoodsList().add(new Goods("挂面", 1.5));
66 ShopManager.addShop(myShop);
67 @Override
68 public String toString() {
69 StringBuffer sb = new StringBuffer();
70 sb.append("name:[" + this.getName() + "]goods:[" +
71 for (Goods goods : getGoodsList()) {
72 sb.append(goods.getName() + "/" +
73 ShopManager goods.getPrice() + ",");
74 }
75 // 判断最后一个字符是否是",",是就删除
76 if (sb.charAt(sb.length() - 1) == ',') {
77 sb.deleteCharAt(sb.length() - 1);
78 }
79 sb.append("]");
80 return sb.toString();
81 }
82 }

```

商店的管理类提供了管理商店的全部功能，包括新增、删除、查找和展示信息的功能：

代码片段15 ShopManager.java

```

1 package cn.steven.pattern.demo.prototype.quest;
2
3 import java.util.Map;
4 import java.util.TreeMap;
5
6 /**
7 * 商店的工具类
8 */
9 public class ShopManager {
10 /**
11 * 用于保存现有的店铺，用于查询和克隆

```

```

12  */
13  private static Map<String, Shop> shopMap =
14      new TreeMap<String, Shop>();
15
16  /**
17   * 查找店铺
18   * @param name 店铺名
19   * @return 店铺
20   */
21  public static Shop findShop(String name) {
22      return shopMap.get(name);
23  }
24
25  /**
26   * 加入连锁店
27   * @param shop
28   */
29  public static void addShop(Shop shop) {
30      shopMap.put(shop.getName(), shop);
31  }
32
33  /**
34   * 删除店铺
35   * @param name 店铺名
36   */
37  public static void removeShop(String name){
38      shopMap.remove(name);
39  }
40
41  /**
42   * 显示所有店铺信息
43   */
44  public static void showAllShop(){
45      for (Map.Entry<String, Shop> entry : shopMap.entrySet()) {
46          System.out.println(entry.getValue());
47      }
48      System.out.println();
49  }
50  }

```

客户端代码:

代码片段16 Client.java

```

1  package cn.steven.pattern.demo.prototype.quest;
2
3  /**
4   * 客户端代码
5   */
6  public class Client {
7
8      public static void main(String[] args) {
9
10
11

```



```

9
10 // 生成总店
11 Shop shop = new Shop();
12 shop.setName("DPC连锁总店");
13 shop.getGoodsList().add(new Goods("鱼子酱", 299.99));
14 shop.getGoodsList().add(new Goods("葡萄酒", 500.5));
15 shop.getGoodsList().add(new Goods("挂面", 5.9));
16
17 // 增加店铺
18 ShopManager.addShop(shop);
19
20 // 显示所有店铺信息
21 ShopManager.showAllShop();
22
23 // 复制店铺
24 Shop myShop = (Shop)ShopManager
25     .findShop("DPC连锁总店").clone();
26 myShop.init("DPC连锁抓哇店");
27 myShop.getGoodsList().add(new Goods("二锅头", 4.9));
28 ShopManager.addShop(myShop);
29
30 Shop myShop2 = (Shop)ShopManager
31     .findShop("DPC连锁抓哇店").clone();
32 myShop2.init("DPC连锁模式店");
33 myShop2.getGoodsList().clear();
34 myShop2.getGoodsList().add(new Goods("龙虾", 900.67));
35 ShopManager.addShop(myShop2);
36
37 // 显示所有店铺信息
38 ShopManager.showAllShop();
39 }
40
41 }

```

注意代码片段16所展示的功能是新增一个Shop对象，然后基于这个对象进行克隆和修改来生成很多对象，这样就极大地方便了Shop对象的创建。

运行结果如图7-11所示。

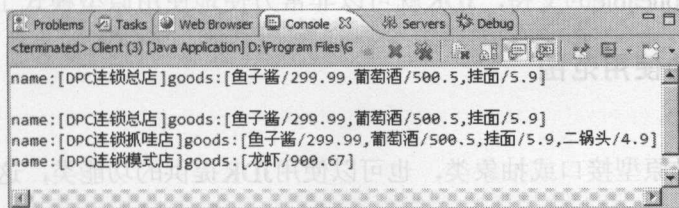


图7-11 运行结果

至此，使用原型模式进行连锁店创建的问题已经解决，在编程过程中可见，如果利用各种工厂模式来解决此问题的话将会麻烦多了，因为商店对象中复杂的属性是不方便在对象新生成时就设计好的。

7.2.4 原型模式在JDK中的应用实例

在编程时，如果想要在Java中实现原型模式，最方便的方法就是使用默认类继承的父类Object，并同时实现Cloneable接口，Cloneable的源代码在代码片段12中已给出，下面给出的就是Object.java中有关克隆部分的代码：

代码片段17 Object.java部分代码

```
1  /*
2   *  @(#)Object.java      1.73 06/03/30
3   *
4   *  Copyright 2006 Sun Microsystems, Inc. All rights reserved.
5   *  SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
6   */
7
8  package java.lang;
9
10 /**
11  * Class <code>Object</code> is the root of the class hierarchy.
12  * Every class has <code>Object</code> as a superclass. All objects,
13  * including arrays, implement the methods of this class.
14  *
15  * @author  unascribed
16  * @version 1.73, 03/30/06
17  * @see    java.lang.Class
18  * @since   JDK1.0
19  */
20 public class Object {
21     ...
22     protected native Object clone() throws CloneNotSupportedException;
23     ...
24 }
```

代码中声明的clone方法是protected属性的，说明此方法只能被其子类直接使用，无法由非子类之外的类调用，同时此方法还是native的，说明这是一个本地实现，通常由虚拟机直接支持此类功能。

有了Object和Cloneable的支持，JDK就可以非常方便地使用原型模式了。

7.2.5 原型模式的使用范围

关键步骤：

- 定义最基本的原型接口或抽象类，也可以使用JDK提供的功能类，这里可以定义一种或多种不同的原型。

- 实现或继承已有的原型接口，定义实体原型对象类，根据需要进行深克隆或浅克隆。
- 创建一个Client性质的类，通过原型模式来加工生产实际的产品对象。

使用范围：

- 当要实例化的类是在运行时刻指定时，例如，通过动态载入。
- 当类实例只是少数不同组合状态的其中之一时，这时比较好的方式是在适当的状态下使

第8章 单例模式 (Singleton)

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类¹。

由定义可以总结出单例模式的三个要点：一是单例类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。

再比如计算机上有多种程序需要播放声音（视频播放器、音频播放器、游戏和操作系统），它们可以同时运行，但是播放声音时却使用同一块声卡进行混音，这时就需要保证声卡是全系统唯一的。

每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免出现不同的实例。

在生活中有时生成对象的方法是这样的，比如在教室中教师和学生需要在黑板上写字，很显然教室中的黑板只有一块，是教师和学生共用的，这种情况下就必须保证取得的黑板是一个共享的唯一的对象。

8.1 连锁店总店的唯一性问题

随着使用原型模式创造的连锁店越来越多，总店的任务就越来越忙了，因为有很多事情是总店要做，而其他的连锁店不用做的，比如新连锁店的注册，加盟费的收取，等等。

根据通常的设计，若想要使用总店对象，需要先创建一个对象，由于超市系统中很多地方都要用到总店对象，所以将会出现很多的创建对象代码。

代码如下所示：

代码片段1 MainShop.java

```
1 package cn.steven.pattern.demo.singleton.quest;
2
3 /**
4  * 连锁店
5  */
6 public class MainShop {
7
8     private String name = "总店";
```

¹Ensure a class only has one instance, and provide a global point of access to it. GOF[95]

```
9
10 public String getName() {
11     return name;
12 }
13
14 public void setName(String name) {
15     this.name = name;
16 }
17
18 }
```

使用总店的代码:

代码片段2 Client.java

```
1 package cn.steven.pattern.demo.singleton.quest;
2
3 /**
4  * 客户代码
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         // 需要使用总店
11         MainShop s = new MainShop();
12         System.out.println(s.getName());
13
14         // 代码中另一个地方需要使用总店
15         MainShop s2 = new MainShop();
16         System.out.println(s.getName());
17
18         System.out.println("是否为同一个店铺:" + (s == s2));
19     }
20
21 }
```

结果如图8-1所示。

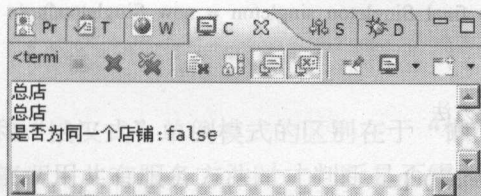


图8-1 Client.java的运行结果

结合代码和运行结果图可见, 在客户使用总店的时候, 如果使用新建对象的方式, 则会导致对象的控制发生混乱。试想如果客户需要向总店发送报表, 使用新建对象的方式, 系统中会出现很多总店的对象, 各种操作就无法统一控制了。

如何对对象进行唯一性的控制, 这就是单例模式需要解决的问题。

8.2 单例模式的结构

单例模式向系统外界提供了唯一的自身对象，外界无法自行创建一个单例的对象。

8.2.1 单例模式

单例模式的类图十分简单，如图8-2所示。

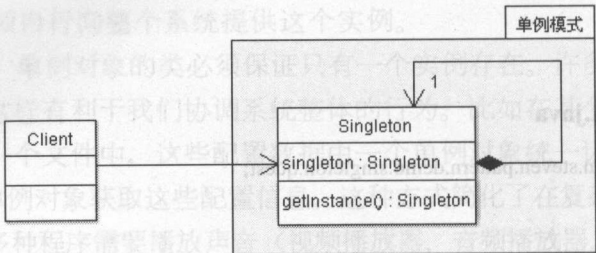


图8-2 单例模式类图

图8-2中Singleton即为一个单例类，它使用组合的方式在类内部创建了一个自身的对象，并通过getInstance方法向外部提供服务。

单例模式的具体实现通常分为“饿汉式”和“懒汉式”，下面分别展示它们的代码。

饿汉式单例模式：

代码片段3 Singleton.java

```
1 package cn.steven.pattern.demo.singleton.pattern;
2
3 /**
4  * 饿汉式单例模式
5  */
6 public class Singleton {
7
8     /**
9      * 私有的静态的自身类对象
10     */
11     private static final Singleton singleton = new Singleton();
12
13     /**
14      * 私有构造方法
15     */
16     private Singleton() {
17
18     }
19
20     /**
21      * 共有的静态获得实例方法
22     */
23     public static Singleton getInstance() {
24         return singleton;
```



```
25 }  
26 }
```

“饿汉式”的特点是:

- 具有私有的构造方法。
- 具有私有的静态属性, 维护自身的实例, 在初始化的时候就创建实例。
- 具有共有的服务方法。

下面展示的是“懒汉式”单例模式:

代码片段4 SingletonLazyload.java

```
1 package cn.steven.pattern.demo.singleton.pattern;  
2  
3 /**  
4  * 单例模式, 懒汉式  
5  */  
6 public class SingletonLazyload {  
7  
8     /**  
9      * 私有的静态的自身类对象  
10     */  
11     private static SingletonLazyload singleton;  
12  
13     /**  
14      * 私有的构造方法  
15     */  
16     private SingletonLazyload() {  
17  
18     }  
19  
20     /**  
21      * 共有的静态获得实例方法  
22     */  
23     public static SingletonLazyload getInstance() {  
24         if(singleton == null){  
25             singleton = new SingletonLazyload();  
26         }  
27         return singleton;  
28     }  
29 }
```

“懒汉式”单例模式和“饿汉式”单例模式的区别在于“懒汉式”不会在类初始化的时候就实例化其对象, 而在外部调用共有服务方法时才判断是否需要进行创建。

通常使用“懒汉式”单例模式的多一些, 因为在初始化的过程中, 它的系统少, 并且在系统运行时, 会有选择地进行实例化工作。

单例模式的使用情况通常有两种:

- 直接新建单例对象。一般我们会加入一个private或者protected的构造函数, 这样系统就不会自动添加public的构造函数了, 因此只能调用里面的static方法, 无法通过new来创建对象。
- 通过反射构造单例对象。反射时可以使用setAccessible方法来突破private的限制, 我们需

要做到直接新建单例对象的同时，在`ReflectPermission("suppressAccessChecks")`权限下使用安全管理器（`SecurityManager`）的`checkPermission`方法来限制这种突破。一般来说，这些事情都是通过应用服务器进行后台配置实现。

如果单例对象有必要实现`Serializable`接口（很少出现），则应当同时实现`readResolve()`方法来保证反序列化的时候得到原来的对象。

8.2.2 多线程情况下的单例模式

在多线程的环境中，简易的单例模式将会出现问题。试想有两个线程进入了代码片段4 `SingletonLazyload.java`试图访问其`getInstance`方法，具体步骤如下：

- 1) 设线程A到达代码23行，则其必然先到达24行。
- 2) 线程B比线程A晚一行代码的速度紧随其后。
- 3) 线程A运行24行之后判断为真（即`singleton`为`null`）于是执行25行。
- 4) 在线程A没有结束25行的运行时，线程B也运行了24行，发现为真（此时线程A并没有执行完25行，所以`singleton`还为`null`）。

5) 于是线程B还会运行25行，此时就会发现在JVM中出现了两个`Singleton`的实例的情况。根据单例模式的定义，就发生错误了。

解决此错误的最简单的方法是使用方法中的`synchronized`关键字。此关键字的含义如下¹：

Synchronized（同步）

Java编程语言提供了两个基本同步语法：同步方法和同步声明。要运用同步的方法，只需添加同步关键字的声明：

```
1 public class SynchronizedCounter {
2     private int c = 0;
3
4     public synchronized void increment() {
5         c++;
6     }
7
8     public synchronized void decrement() {
9         c--;
10    }
11
12    private static final SynchronizedCounter singleton = new SynchronizedCounter();
13
14    public synchronized int value() {
15        return c;
16    }
17 }
```

`SynchronizedCounter`类的实例中有三个同步的方法，使这些方法同步有两个作用：

- 第一，同步的方法不能被两个线程同时访问。当一个线程执行一个同步方法块时，所有其他调用此同步块的方法暂停，直到第一个线程执行完毕。
- 第二，在执行线程从同步方法退出时，JVM将会在阻塞的线程中选择一个进入此方

¹<http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html>.

法，其他线程依然等待，这保证了所有使用此方法的线程会被依次调用。

使用同步方法的代码如下：

代码片段5 SingletonLazyloadSyn.java

```
1 package cn.steven.pattern.demo.singleton.pattern;
2
3 /**
4  * 单例模式，懒汉式，方法同步
5  */
6 public class SingletonLazyloadSyn {
7
8     /**
9      * 私有的静态的自身类对象
10     */
11     private static SingletonLazyloadSyn singleton;
12
13     /**
14      * 私有构造方法
15     */
16     private SingletonLazyloadSyn() {
17
18     }
19
20     /**
21      * 共有的静态获得实例方法
22     */
23     public static synchronized SingletonLazyloadSyn getInstance() {
24         if (singleton == null) {
25             singleton = new SingletonLazyloadSyn();
26         }
27         return singleton;
28     }
29 }
```

代码片段5运行步骤如下：

1) 设线程A首先到达代码23行，检查同步对象的状态，发现没有加锁，于是将对象加锁后到达24行。

2) 线程B以比线程A晚一行代码的速度紧随其后。当其到达23行时，检查同步对象的状态（这两个线程使用同一个同步对象），发现已经加锁，于是线程阻塞。

3) 线程A运行24行之后判断为真（即singleton为null），于是执行25行并随后结束了方法的调用。

4) 线程A结束方法调用后会将同步对象解锁，并通知所有等待此锁的线程争用。

5) 线程B获得了操作权，对同步对象加锁后顺利执行了方法并解锁。

使用的例子如下：

代码片段6 ClientSingletonLazyloadSyn.java

```
1 package cn.steven.pattern.demo.singleton.pattern;
2
3 /**
4  * 测试多线程
5  */
6 public class ClientSingletonLazyloadSyn {
7     public static void main(String[] args) throws InterruptedException {
8
9         final ThreadPair tp = new ThreadPair();
10
11         /**
12          * 创建两个线程
13          */
14         Thread t1 = new Thread(new Runnable() {
15             @Override
16             public void run() {
17                 tp.setS1(SingletonLazyloadSyn.getInstance());
18             }
19         });
20
21         Thread t2 = new Thread(new Runnable() {
22             @Override
23             public void run() {
24                 tp.setS2(SingletonLazyloadSyn.getInstance());
25             }
26         });
27
28         /**
29          * 启动线程
30          */
31         t1.start();
32         t2.start();
33
34         /**
35          * 加入线程等待
36          */
37         t1.join();
38         t2.join();
39
40         System.out.println("相同的对象？" + tp.isEquals());
41     }
42 }
43
44 class ThreadPair {
45     SingletonLazyloadSyn s1;
46     SingletonLazyloadSyn s2;
47
48     public SingletonLazyloadSyn getS1() {
49         return s1;
50     }
```

```
51 synchronized (SingletonLazyloadSyn.class) {
52     public void setS1(SingletonLazyloadSyn s1) {
53         this.s1 = s1;
54     }
55
56     public SingletonLazyloadSyn getS2() {
57         return s2;
58     }
59
60     public void setS2(SingletonLazyloadSyn s2) {
61         this.s2 = s2;
62     }
63
64     /**
65      * 判断是否为同一个对象
66      */
67     public boolean isEqual() {
68         return s1 == s2;
69     }
70 }
```

注意代码片段6中模拟了两个线程同时访问单例类的情况，运行结果如图8-3所示。

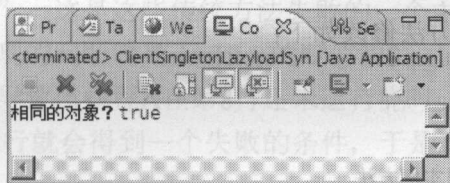


图8-3 同步方法运行结果

所以，使用同步方法可以解决多线程问题，代码片段5所示的单例模式的编程方式是推荐使用的方式之一。

另一种能保证线程安全的方法是《Effective Java》一书中给出的一种精妙的解决方法，它充分利用了Java虚拟机的特性。

代码片段7 SingletonLazyloadSynSafe.java

```
1 package cn.steven.pattern.demo.singleton.pattern;
2
3 /**
4  * 单例模式，线程安全
5  */
6 public class SingletonLazyloadSynSafe {
7
8     // 内部类
9     private static class SingletonHolder {
10         static final SingletonLazyloadSynSafe uniqueInstance
11             = new SingletonLazyloadSynSafe();
12     }
13
14     /**
```

```

15     * 私有构造方法
16     */
17     private SingletonLazyloadSynSafe() {
18
19     }
20
21     /**
22     * 共有的静态获得实例方法
23     */
24     public static SingletonLazyloadSynSafe getInstance() {
25         return SingletonHolder.uniqueInstance;
26     }
27 }

```

8.2.3 单例模式的双重检查模式DCL

在学习双重检查模式前，请再次查看代码片段5，注意此类中公有的`getInstance`方法和步骤的描述。

我们可以得出结论，这种多线程的设计虽然可以避免多线程问题，但是使用这个方法效率是很低下的，因为同一时间内，只能有一个线程进入方法内部，那么是否可以提高一下效率呢？

下面将其中的`getInstance`方法修改如下：

代码片段8 DCL_1

```

1  /**
2   * 共有的静态获得实例方法
3   */
4   public static SingletonLazyloadSyn getInstance() {
5       if (singleton == null) {
6           synchronized (SingletonLazyloadSyn.class) {
7               singleton = new SingletonLazyloadSyn();
8           }
9       }
10      return singleton;
11  }

```

注意，代码中将放在方法上的同步声明转移到了内部的代码块中，这样就不是所有的线程都需要访问锁机制了，只有当`singleton`为`null`的时候才会使用锁机制。但是问题在于假设有两个线程同时到达第6行，很显然会一个继续运行，另一个等待，但是最终的结果还是运行了两遍第7行，这并不符合要求。

可以进一步修改如下：

代码片段9 DCL_2

```

1  /**
2   * 共有的静态获得实例方法
3   */
4   public static SingletonLazyloadSyn getInstance() {
5       if (singleton == null) {

```



```

6      synchronized (SingletonLazyloadSyn.class) {
7          if (singleton == null) {
8              singleton = new SingletonLazyloadSyn();
9          }
10     }
11 }
12 return singleton;
13 }

```

代码修改之后增加了第7行代码，可见第7行和第5行是完全相同的代码，这就是典型的双重检查 (Double Checked Locking¹)，其典型步骤为：

- 1) 检查变量是否被初始化（不用锁），如果已经被初始化，立即返回此变量。
- 2) 获得锁。
- 3) 再次检查变量是否被初始化，如果变量已被前一个线程初始化，则不操作，返回变量。
- 4) 否则初始化变量并返回。

很遗憾，传统的双重检查在Java中是不能工作的（在C#中同样也不能正常工作）。双重检查锁定的理论是完美的，不幸地是，现实完全不同。双重检查锁定的问题是：并不能保证它会在单处理器或多处理器计算机上顺利运行。

双重检查锁定的失败并不归咎于JVM中的实现bug，而是归咎于Java平台内存模型。内存模型允许所谓的“无序写入”，这是这些传统方法失败的一个主要原因²。

“无序写入”在这里出现的问题就是singleton的赋值和SingletonLazyloadSyn的初始化的顺序是不确定的，所以有可能导致当线程A在第8行还未运行完时singleton的值就不为null了，所以此时线程B如果执行到第5行就会得到一个失败的条件，于是返回一个没有被成功初始化的singleton。

这个问题的解决方案有很多种，使用“饿汉式”的方式可以解决此问题（代码片段3），因为在多线程的情况下JVM可以保证类中的静态内容只被初始化一次。

建议以代码片段7的方式使用“懒汉式”的单例模式。如果非要使用DCL模式，需要使用volatile关键字（JDK1.5+参见JSR133³新内存模型⁴），此关键字的意图为：JDK5+通过定义告知虚拟机被volatile修饰的变量的读写不允许被排序⁵。

实现代码如下：

代码片段10 SingletonLazyloadDCL.java

```

1 package cn.steven.pattern.demo.singleton.pattern;
2
3 import java.util.concurrent.locks.ReentrantLock;
4

```

¹http://en.wikipedia.org/wiki/Double-checked_locking.

²<http://www.ibm.com/developerworks/cn/java/j-dcl.html>.

³<http://jcp.org/en/jsr/detail?id=133>.

⁴<http://www.cs.umd.edu/~pugh/java/memoryModel/>.

⁵JDK5 and later extends the semantics for volatile so that the system will not allow a write of a volatile to be reordered with respect to any previous read or write, and a read of a volatile cannot be reordered with respect to any following read or write.

```
5  /**
6   * 单例模式；懒加载；错误的双检查
7   */
8  public class SingletonLazyloadDCL {
9
10     /**
11      * 私有静态的自身类对象
12      * 需定义为volatile特性（需JDK1.5以上版本此关键字才会被发现）
13      */
14     private static volatile SingletonLazyloadDCL singleton;
15
16     /**
17      * 私有构造方法
18      */
19     private SingletonLazyloadDCL() {
20
21     }
22
23     /**
24      * 共有的静态获得实例方法
25      */
26     public static SingletonLazyloadDCL getInstance() {
27         if (singleton == null) {
28             /**
29              * 使用此类对象进行同步
30              * 注意singleton必需定义为volatile特性（需JDK1.5以上版本此关键字才会被发现）
31              */
32             synchronized (SingletonLazyloadDCL.class) {
33                 if (singleton == null) {
34                     singleton = new SingletonLazyloadDCL();
35                 }
36             }
37         }
38     }
39     return singleton;
40 }
41 }
```

如果使用互斥锁，也可以实现DCL：

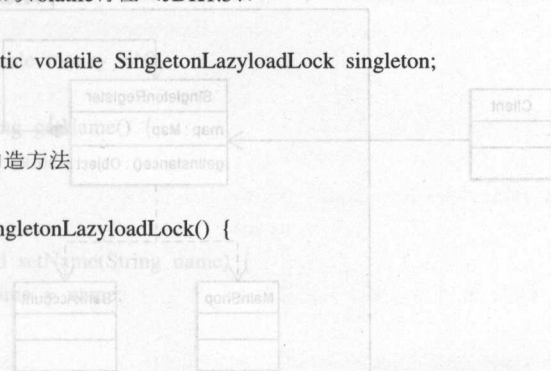
代码片段11 SingletonLazyloadLock.java

```
1  package cn.steven.pattern.demo.singleton.pattern;
2
3  import java.util.concurrent.locks.ReentrantLock;
4
5  /**
6   * 单例模式；懒加载；互斥锁
7   */
8  public class SingletonLazyloadLock {
9
10     /**
```

```

11     * 互斥锁
12     */
13     private static final ReentrantLock lock = new ReentrantLock();
14
15     /**
16      * 私有静态的自身类对象
17      * 需定义为volatile特性 (JDK1.5+)
18      */
19     private static volatile SingletonLazyloadLock singleton;
20
21     /**
22      * 私有构造方法
23      */
24     private SingletonLazyloadLock() {
25     }
26
27     /**
28      * 共有的静态获得实例方法
29      */
30     public static SingletonLazyloadLock getInstance() {
31         if (singleton == null) {
32             /**
33              * 锁定访问线程
34              * 注意singleton必须定义为volatile特性 (JDK1.5+)
35              */
36             lock.lock();
37             try {
38                 if (singleton == null) {
39                     singleton = new SingletonLazyloadLock();
40                 }
41             } finally {
42                 /**
43                  * 解锁
44                  */
45                 lock.unlock();
46             }
47             return singleton;
48         }
49     }
50 }

```



至此，双重检查的问题已经讨论结束，在C/C++中使用的模式并不一定能用于Java/C#这种虚拟机语言，因为虚拟机特性会有可能优化其代码的运行方式。

多线程中使用单例模式的建议是不要使用任何双重检查模式，直接使用代码片段3中的饿汉式或代码片段7中的懒汉式即可。

8.2.4 使用单例模式解决连锁店总店的问题

前面几节讲述的单例模式是非常有效的，但是单例模式还存在几个问题：

- 单例化的类无法扩展其子类。

- 将一个类实现为单例类是一种侵入性的编程方法。
- 如果要动态载入类为单例不可行。

有一种方法是用注册的方式管理单例类的，这对上面的三个问题有了很好的解决方法，新连锁店总店的设计可以采用此方法，下面是设计类图，如图8-4所示。

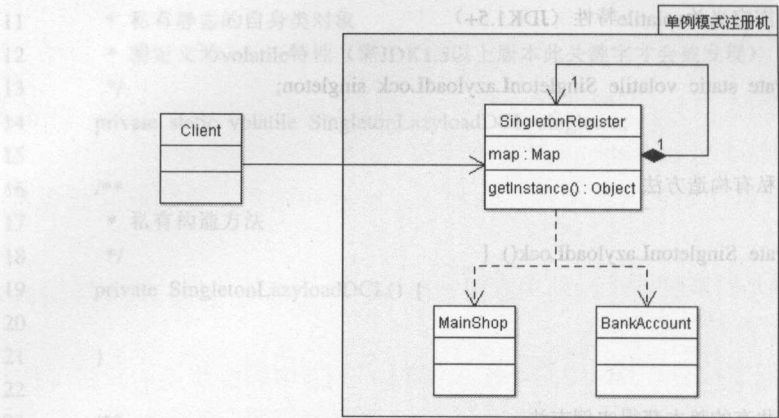


图8-4 单例模式注册机类图设计

图8-4中把MainShop和BankAccount两个类注册为单例类。SingletonRegister类的作用是注册和维护所有的单例类。

下面首先展示的是需要单例化的类。

代码片段12 MainShop.java

```
1 package cn.steven.pattern.demo.singleton;
2
3 /**
4  * 连锁店总店，是唯一的
5  */
6 public class MainShop {
7
8     private String name = "总店";
9
10    public String getName() {
11        return name;
12    }
13
14    public void setName(String name) {
15        this.name = name;
16    }
17 }
```

MainShop类是一个很普通的类，从代码中看不出它将成为一个单例类。

BankAccout类的作用是提供供所有连锁店共享的一个银行账户，因为每个月各个连锁店都要向总店汇款，为了方便管理，共用一个账户是比较合适的。

代码片段13 BankAccount.java

```
1 package cn.steven.pattern.demo.singleton;
```

```

2
3 /**
4  * 共用账户，是唯一的
5  */
6 public class BankAccount {
7
8     private String name = "模式连锁店账户";
9
10    private double money = 0;
11
12    public String getName() {
13        return name;
14    }
15
16    public void setName(String name) {
17        this.name = name;
18    }
19
20    public double getMoney() {
21        return money;
22    }
23
24    /**
25     * 将设置金额的方法设置为同步，防止出现多线程异常
26     * @param money
27     */
28    public synchronized void setMoney(double money) {
29        this.money = money;
30    }
31
32 }

```

下面展示的是这个模式中的核心类——注册机类。

代码片段14 SingletonRegister.java

```

1 package cn.steven.pattern.demo.singleton;
2
3 import java.util.HashMap;
4 import java.util.concurrent.locks.Lock;
5 import java.util.concurrent.locks.ReentrantReadWriteLock;
6
7 /**
8  * 单例模式注册类
9  * 客户需要通过此类获得所有单例对象
10 * 此类本身使用“饿汉式”单例模式
11 */
12 public class SingletonRegister {
13
14     /**
15      * 自身维护的单例模式类
16      */

```

```

17     private static final SingletonRegister s
18         = new SingletonRegister();
19
20     /**
21      * 维护注册信息的Map
22      */
23     private static final HashMap<String, Object> map
24         = new HashMap<String, Object>();
25
26     /**
27      * 处理多线程的可重入读写锁
28      */
29     private static final ReentrantReadWriteLock rwlock
30         = new ReentrantReadWriteLock();
31
32     /**
33      * 只读锁。此锁可以多个只读锁并存，锁定时不可以有可写锁
34      */
35     private static final Lock r = rwlock.readLock();
36
37     /**
38      * 可写锁。此锁必须独占，锁定时不能有其他任何锁
39      */
40     private static final Lock w = rwlock.writeLock();
41
42     /**
43      * 私有构造方法
44      */
45     private SingletonRegister() {
46     }
47
48     /**
49      * 用于取得某个类的单例对象
50      *
51      * @param className
52      *         类名称
53      * @return
54      * @throws ClassNotFoundException 类未找到
55      * @throws IllegalAccessException 无法访问类
56      * @throws InstantiationException 无法实例化
57      */
58     public static Object getInstance(String className)
59         throws InstantiationException, IllegalAccessException,
60             ClassNotFoundException {
61
62         Object c = null;
63
64         /**
65          * 获得读取锁
66          */
67         r.lock();

```



```

68     try {
69         System.out.println("读锁:")
70         +Thread.currentThread().getName());
71         c = map.get(className);
72         if (c != null) {
73             return c;
74         }
75     } finally {
76         /**
77          * 释放读取锁
78          */
79         r.unlock();
80     }
81
82     /**
83      * 获得写锁
84      */
85     w.lock();
86     try {
87         System.out.println("写锁:")
88         +Thread.currentThread().getName());
89         /**
90          * 双重检查
91          */
92         c = map.get(className);
93         if (c != null) {
94             System.out.println("直接使用已注册的对象:")
95             +Thread.currentThread().getName());
96             return c;
97         }
98         /**
99          * 使用反射的方式实例化对象
100          */
101         c = Class.forName(className).newInstance();
102         /**
103          * 放入注册表中
104          */
105         map.put(className, c);
106         System.out.println("新创建的对象:")
107         +Thread.currentThread().getName());
108     } finally {
109         /**
110          * 释放写锁
111          */
112         w.unlock();
113     }
114
115     return c;
116 }
117
118 }

```

注意代码片段14中的类本身就是一个是用了“饿汉式”单例模式的类，为了保证全局注册机的唯一性，它的功能是向外界提供一个所有单例类的管理机制。

为了保证线程安全和能够高效地访问，代码中使用了可重入的读写锁`ReentrantReadWriteLock`，它的功能和特点如下所示：

维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有`Writer`，读取锁可以由多个`Reader`线程同时保持。写入锁是独占的。

所有`ReadWriteLock`实现都必须保证`WriteLock`操作的内存同步效果也要保持与相关`ReadLock`的联系。也就是说，成功获取读锁的线程会看到写入锁之前，版本所做的所有更新。

与互斥锁相比，读写锁允许对共享数据进行更高级别的并发访问。虽然一次只有一个线程（`Writer`线程）可以修改共享数据，但在许多情况下，任何数量的线程都可以同时读取共享数据（`Reader`线程），读写锁利用了这一点。从理论上讲，与互斥锁相比，使用读写锁所增强的并发性将带来更多的性能提高。在实践中，只有在多处理器上并且只在访问模式适用于共享数据时，才能完全实现并发性增强。

与互斥锁相比，使用读写锁能否提升性能则取决于读写操作期间读取数据相对于修改数据的频率，以及数据的争用——即在同一时间试图对该数据执行读取或写入操作的线程数。例如，某个最初用数据填充并且之后不经常对其进行修改的`collection`，因为经常对其进行搜索（比如搜索某种目录），所以这样的`collection`是使用读写锁的理想候选者。但是，如果数据更新变得频繁，数据在大部分时间都被独占锁锁定，这时，就算存在并发性增强，也是微不足道的。更进一步地说，如果读取操作所用时间太短，则读写锁实现（它本身就比互斥锁复杂）的开销将成为主要的执行成本。最终，只有通过分析和测量，才能确定应用程序是否适合使用读写锁。

尽管读写锁的基本操作是直截了当的，但实现时仍然必须做出许多决策，这些决策可能会影响给定应用程序中读写锁的效果。这些策略的例子包括：

- 在`Writer`释放写入锁时，`Reader`和`Writer`都处于等待状态，这时要确定是授予读取锁还是授予写入锁。`Writer`优先比较普遍，因为预期写入所需的时间较短并且不那么频繁。`Reader`优先不太普遍，因为如果`Reader`正如预期的那样频繁和持久，那么它将导致花费相比写入操作来说较长的时延。

- 在`Reader`处于活动状态而`Writer`处于等待状态时，确定是否向请求读取锁的`Reader`授予读取锁。`Reader`优先会无限期地延迟`Writer`，而`Writer`优先会减少可能的并发。

- 确定是否重新进入锁：可以使用带有写入锁的线程重新获取它吗？可以在保持写入锁的同时获取读取锁吗？可以重新进入写入锁本身吗？

- 可以将写入锁在不允许其他`Writer`干涉的情况下降级为读取锁吗？可以优先于其他等待的`Reader`或`Writer`将读取锁升级为写入锁吗？

当评估给定实现是否适合你的应用程序时，应该考虑所有这些情况。

注意代码片段14中第44行使用了反射的方式进行动态类对象的创建。

客户端使用的代码如下所示：

代码片段15 Client.java

```
1 package cn.steven.pattern.demo.singleton;
2
3 /**
```

```

4  * 测试多线程
5  */
6  public class Client {
7      public static void main(String[] args)
8          throws InterruptedException {
9
10         final ThreadPair tp = new ThreadPair();
11
12         /**
13          * 创建两个线程
14          */
15         Thread t1 = new Thread(new Runnable() {
16             @Override
17             public void run() {
18                 try {
19                     tp.setS1(SingletonRegister
20                         .getInstance("cn.steven.pattern.demo.singleton.MainShop"));
21                 } catch (InstantiationException e) {
22                     e.printStackTrace();
23                 } catch (IllegalAccessException e) {
24                     e.printStackTrace();
25                 } catch (ClassNotFoundException e) {
26                     e.printStackTrace();
27                 }
28             }
29         });
30
31         Thread t2 = new Thread(new Runnable() {
32             @Override
33             public void run() {
34                 try {
35                     tp.setS2(SingletonRegister
36                         .getInstance("cn.steven.pattern.demo.singleton.MainShop"));
37                 } catch (InstantiationException e) {
38                     e.printStackTrace();
39                 } catch (IllegalAccessException e) {
40                     e.printStackTrace();
41                 } catch (ClassNotFoundException e) {
42                     e.printStackTrace();
43                 }
44             }
45         });
46
47         /**
48          * 启动线程
49          */
50         t1.setName("线程甲");
51         t2.setName("线程乙");
52         t1.start();
53         t2.start();
54

```



```
55  /**
56  * 加入线程等待
57  */
58  t1.join();
59  t2.join();
60
61  System.out.println("相同的对象?" + tp.isEquals());
62  }
63  }
64
65  class ThreadPair {
66      private Object s1;
67      private Object s2;
68
69      public Object getS1() {
70          return s1;
71      }
72
73      public void setS1(Object s1) {
74          this.s1 = s1;
75      }
76
77      public Object getS2() {
78          return s2;
79      }
80
81      public void setS2(Object s2) {
82          this.s2 = s2;
83      }
84
85      /**
86       * 判断是否为同一个对象
87       */
88      public boolean isEquals() {
89          System.out.println(s1+"\n"+s2);
90          return s1 == s2;
91      }
92  }
```

运行结果如图8-5所示。

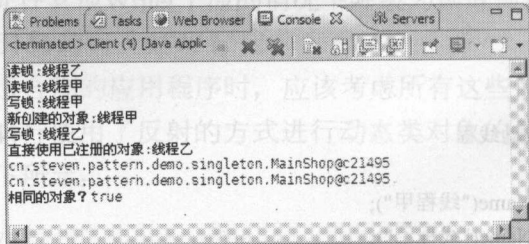


图8-5 运行结果

由结果可见，两个线程得到的对象是同一个对象，使用注册表管理单例类的好处是不要求所管理的单例类是特殊方法编程的，它可以支持继承，而且可以动态加载类。

至此，就已经解决了连锁店总店唯一性的问题了，并且可以支持将多个类单例化。

8.2.5 单例模式在JDK中的实例

单例模式在JDK和各种框架中十分常见，因为在各种需求下都有对某种资源唯一化的需求。

例如java.lang.System，此类提供了很多工具方法，比如获取系统输入和输出对象，获取系统属性，获取系统当前时间等，其典型的用法如下：

代码片段16 Test_1.java

```

1 package cn.steven.pattern.demo.singleton.jdk;
2
3 import java.util.Map;
4 import java.util.Properties;
5
6 /**
7  * 测试单例类
8  */
9 public class Test_1 {
10     public static void main(String[] args) {
11
12         /**
13          * 获得当前时间的毫秒数
14          */
15         long time = System.currentTimeMillis();
16         System.out.println("当前毫秒: "+time);
17
18         /**
19          * 获取系统属性
20          */
21         Properties properties = System.getProperties();
22         for (Map.Entry<Object, Object> e : properties.entrySet()) {
23             System.out.println("「 "+e.getKey()+" 」 「 "+e.getValue()+" 」");
24         }
25
26         /**
27          * 下面的代码是错的，因为System的构造方法为私有属性
28          */
29         //System sys = new System();//error
30     }
31 }

```

部分运行结果如图8-6所示。

由于JDK对外开放了System这个单例，所以系统代码中任何一个地方都可以使用这些服务而无需考虑其对象的多例问题以及多线程安全问题。

再比如JDK中的Runtime类，此类就是一个典型的单例模式，典型用法如下：

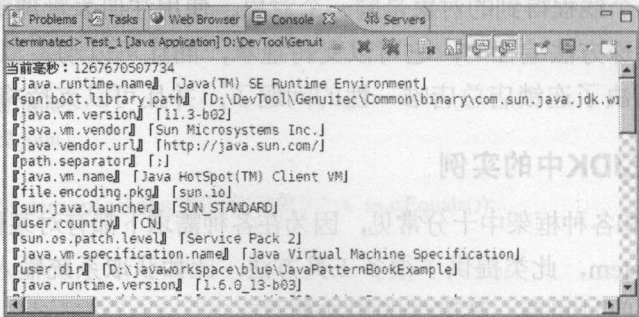


图8-6 部分运行结果

代码片段17 TLTest.java

```
1 package cn.steven.pattern.demo.singleton.jdk;
2
3 /**
4  * 测试Runtime
5  */
6 public class TLTest {
7     public static void main(String[] args) {
8         // Runtime rt = new Runtime();//error
9         Runtime runtime = Runtime.getRuntime();
10        System.out.println("JVM空闲内存为: " + runtime.freeMemory());
11        //result: JVM空闲内存为: 4934320
12    }
13 }
```

JDK中还设计了一个类java.lang.ThreadLocal<T>，此类虽然和前面所说的全局单例无关，但是它的功能是：在多线程情况下，每一个线程所使用的特定实例是单例的，其描述如下：

```
public class ThreadLocal<T>
extends Object
```

该类提供了线程局部（thread-local）变量。这些变量不同于普通的类中属性，访问此类的对象（通过其get或set方法）的每个线程都有自己的局部变量，它独立于变量的初始化副本。此类的实例通常是类中的private static字段，它们希望将其中存放的值与某一个线程（例如，用户ID或事务ID）相关联，这样每一个使用它的线程都拥有了不同于其他线程的变量。

例如，以下类生成对每个线程唯一的局部标识符。线程ID是在第一次调用UniqueThreadIdGenerator.getCurrentThreadId()时分配的，在后续调用中不会更改。

```
1 import java.util.concurrent.atomic.AtomicInteger;
2
3 public class UniqueThreadIdGenerator {
4
5     private static final AtomicInteger uniqueId = new AtomicInteger(0);
6
7     private static final ThreadLocal < Integer > uniqueNum =
8         new ThreadLocal < Integer > () {
```



```

9         @Override protected Integer initialValue() {
10             return uniqueId.getAndIncrement();
11         }
12     };
13
14     public static int getThreadId() {
15         return uniqueId.get();
16     }
17 } // UniqueThreadIdGenerator

```

每个线程都保持对其线程局部变量副本的隐式引用，只要线程是活动的并且 ThreadLocal 实例是可访问的；在线程消失之后，其线程局部实例的所有副本都会被作为垃圾回收（除非存在对这些副本的其他引用）。

从以下版本开始：JDK1.2

在利用 Hibernate 框架开发 DAO 模块时，我们和 Session 接触得最多，所以如何合理管理 Session，避免 Session 的频繁创建和销毁，对于提高系统的性能来说是非常重要的，以下代码实现了 Session 管理功能。

代码片段18 HibernateSessionFactory.java

```

1 import org.hibernate.HibernateException;
2 import org.hibernate.Session;
3 import org.hibernate.cfg.Configuration;
4
5 public class HibernateSessionFactory {
6
7     private static String CONFIG_FILE_LOCATION = "/hibernate.cfg.xml";
8     private static final ThreadLocal threadLocal = new ThreadLocal();
9     private static Configuration configuration = new Configuration();
10    private static org.hibernate.SessionFactory sessionFactory;
11    private static String configFile = CONFIG_FILE_LOCATION;
12
13    static {
14        try {
15            configuration.configure(configFile);
16            sessionFactory = configuration.buildSessionFactory();
17        } catch (Exception e) {
18            System.err.println("创建失败");
19            e.printStackTrace();
20        }
21    }
22
23    private HibernateSessionFactory() {
24        //单例工厂
25    }
26
27    public static Session getSession() throws HibernateException {
28
29        Session session = (Session) threadLocal.get();
30        if (session == null || !session.isOpen()) {

```

```
31         if (sessionFactory == null) {
32             rebuildSessionFactory();
33         }
34         session = (sessionFactory != null) ?
35             sessionFactory.openSession(): null;
36         threadLocal.set(session);
37     }
38     return session;
39 }
40 // Other methods...
41 }
```

Session是由SessionFactory负责创建的，而SessionFactory的实现是具有线程安全的，采用了前面提到的“饿汉式”创建单例。多个并发的线程可以同时访问一个SessionFactory并从中获取Session实例，那么Session是否是线程安全的呢？很遗憾，答案是否定的。Session中包含了与数据库操作相关的状态信息，那么说如果多个线程同时使用一个Session实例进行CRUD，就很有可能导致数据存取的混乱，你能够想象那些你根本不能预测执行顺序的线程对你的一条记录进行操作的情形吗？代码片段18使用ThreadLocal模式解决了这一问题。只要借助上面的工具类获取Session实例，我们就可以实现线程范围内的Session共享，从而避免了线程中频繁创建和销毁Session实例。当然，不要忘记在用完后关闭Session。

JDK和各种框架中有大量的单例模式的例子，读者可以在学习过程中进行积累总结。

8.2.6 单例模式的使用范围

单例模式的功能：

- 确保一个类只有一个实例被建立。
- 提供了一个对对象的全局访问指针。
- 在不影响单例类的客户端的情况下允许将来有多个实例。

单例模式的优点：

为一个面向对象的应用程序提供了对象的唯一访问点，不管它实现何种功能，此种模式都为设计及开发团队提供了共享的概念。

单例模式的缺点：

单例模式类的派生有很大的困难，因为其构造方法为私有，所以子类也不能使用。

单例模式的应用情景：

系统只需要一个实例对象，或客户调用类的单个实例只允许使用一个公共访问点时。

8.2.7 与其他模式的关系

通常来说单例模式的特点非常明显，不会和其他的模式混淆，有时它和原型模式会记忆混淆，注意其定义如下：

- 单例模式（Singleton）：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- 原型模式（Prototype）：用原型实例指定创建对象的种类，并且通过拷贝这个原型创建新的对象。

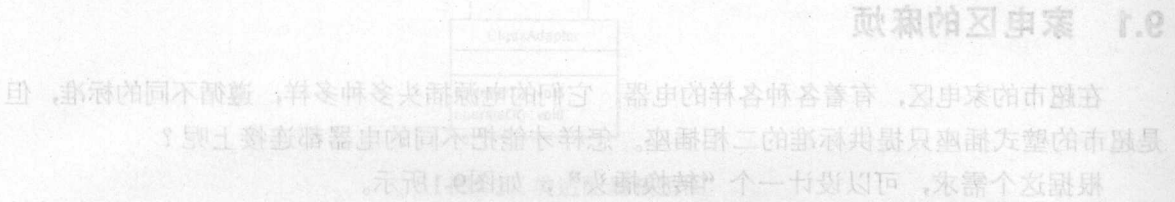
注意他们创建对象的方法和目的都不相同。

8.3 单例模式总结

单例模式简单却容易让人迷惑，特别是对于Java程序员，存在特别多的注意事项。学习单例模式时要循序渐进，充分理解多线程情况下单例的状态。

9.2.1 类适配器

在现实生活中，我们经常会遇到这样的情况：比如，我们有一个接口，它定义了一些方法，我们有一个类，它实现了这个接口，但是，这个类的方法实现与接口的方法实现不一致，我们想要在不改变这个类的情况下，让这个类的方法实现与接口的方法实现一致，这就是类适配器的作用。类适配器是一个类，它实现了接口，并且它的方法实现与接口的方法实现一致。类适配器通常用于将一个类适配到一个接口，或者将一个类适配到另一个类。



原有系统中的接口和类无法满足客户需求，所以我们需要设计一个类ClassAdapter，它继承了原有系统中的OneAdapter，再使此实现客户接口Target，这样，就可以在保留原有系统功能的同时实现新接口的功能。

注意，通常设计类适配器的时候全用类重载的方法，可在原类的基础上进行修改，如果方法完全不能用，就再重新设计新方法。

参考代码如下：

代码片段1：OneAdapter源代码

```
3 * 适配器接口
4
5 public interface OneAdapter {
6     //方法A
7     public void methodA();
8     //方法B
9     public void methodB();
10 }
```

代码片段2：OneAdapter实现类
package cn.steven.pattern.adapter;
/**
 * Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. (GoF1995)
 * 注意：此处的接口并不是指接口中的接口。
 */

第9章 适配器模式 (Adapter)

适配器模式在编程过程中是一个经常用到的模式，它的作用是将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作¹。

在生活中，这样的例子也屡见不鲜，比如在给手机充电的时候，不可能直接在220V电源上充电，而是通过手机“充电器”转换成手机适用的电压才可以正常充电，否则就不可以充电，这个“充电器”就起到了适配器的作用。再如，现在的螺丝有很多种，有扁口的、梅花口的、内六方口的、外六方口的……，过去单一的螺丝刀已不能操作这些花样众多的螺丝。后来出现了螺丝刀套装，一个手柄，多个不同的刀头，这样用一个手柄就可以操作多种螺丝了。新的电脑鼠标一般都是USB接口，而旧的电脑机箱上根本就没有USB接口，只有一个PS2接口，这时就必须有一个PS2转USB的适配器，将PS2接口适配为USB接口。

9.1 家电区的麻烦

在超市的家电区，有着各种各样的电器，它们的电源插头多种多样，遵循不同的标准，但是超市的壁式插座只提供标准的二相插座。怎样才能把不同的电器都连接上呢？

根据这个需求，可以设计一个“转换插头”，如图9-1所示。

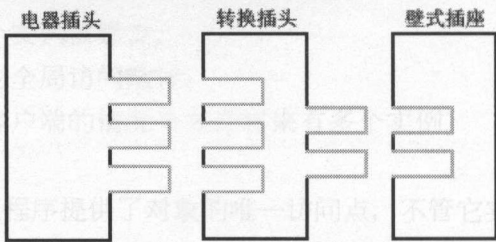


图9-1 电源插头转换图示

由图9-1可以看出，只需要设计一个转换插头，本来不能结合在一起的部件就可以结合使用了。这也正是适配器模式在软件设计范畴所要解决的问题。在此例中，图9-1中的三者有各自的名字：

- 壁式插座：目标 (Target)，是需要转换成的接口²。
- 电器插头：源 (Adaptee)，是需要转换的接口。
- 转换插头：适配器 (Adapter)，是用来转换的接口。

思考：

- a) 还有其他解决插座问题的方法吗？
- b) 如果电压不一样应该如何解决？

¹Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. GOF[95]
²注意此处的接口并不是指软件中的接口。

9.2 适配器的结构

适配器模式提供了具有所需接口的包装类。
常见的适配器模式的结构有类适配器和对象适配器两种形式，它们的主要区别在于类适配器采用继承的方式使用源，而对象适配器采用委托的方式使用源。

9.2.1 类适配器

类适配器将源API转换成目标API的UML静态模型图如图9-2所示。

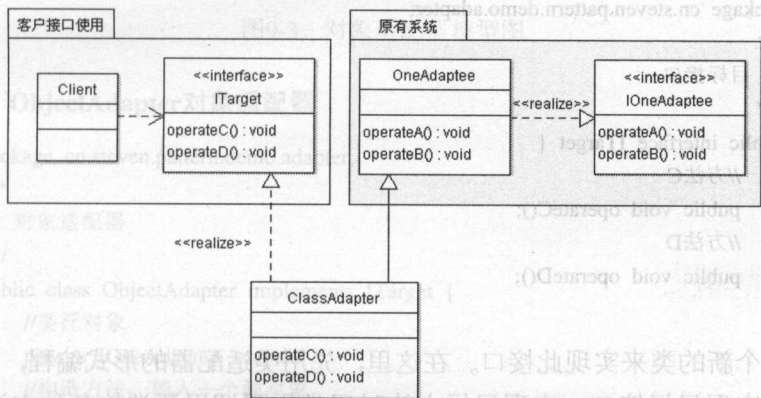


图9-2 类适配器模型图

原有系统中的接口和类并不能满足客户的要求，所以新建一个类ClassAdapter，它继承了原有系统中的OneAdaptee，再使此类实现客户接口ITarget，这样，就可以在保留原有系统功能的同时实现新接口的功能了。

注意，通常设计类适配器的时候会用到源类的方法，可在源类的基础上进行修改，如果方法完全不能用，就再重新设计新方法。

参考代码如下：

代码片段1 IOneAdaptee源代码

```
1 package cn.steven.pattern.demo.adapter;
2 /**
3  * 源所实现的接口
4  */
5 public interface IOneAdaptee {
6     //方法A
7     public void operateA();
8     //方法B
9     public void operateB();
10 }
```

代码片段2 OneAdaptee源代码

```
1 package cn.steven.pattern.demo.adapter;
2 /**
```

```
3  * OneAdaptee 一个源
4  */
5  public class OneAdaptee implements IOneAdaptee {
6      public void operateA() {
7      }
8      public void operateB() {
9      }
10 }
```

代码片段1和代码片段2展示了原有系统的功能，可以发现原有的代码并不能满足目标接口。

代码片段3 目标接口ITarget

```
1  package cn.steven.pattern.demo.adapter;
2  /**
3   * 目标接口
4   */
5  public interface ITarget {
6      //方法C
7      public void operateC();
8      //方法D
9      public void operateD();
10 }
```

下面设计一个新的类来实现此接口。在这里，先用类适配器的形式编程，要点是生成一个源类的子类，并实现目标接口，实现目标方法时通常需要调用源类的各种方法。

代码片段4 ClassAdapter类适配器

```
1  package cn.steven.pattern.demo.adapter;
2  /**
3   * 类适配器
4   */
5  public class ClassAdapter extends OneAdaptee implements ITarget {
6      //目标接口中的方法
7      public void operateC() {
8          //在此可使用父类的各种方法
9      }
10     //目标接口中的方法
11     public void operateD() {
12         //在此可使用父类的各种方法
13     }
14 }
```

9.2.2 对象适配器

对象适配器将源API转换成目标API的UML静态模型图，如图9-3所示。

请仔细对照图9-2与图9-3，看出差别来了吗？类适配器是用继承的方式使用源类，而对象适配器是使用委托的方式使用源类的对象。这也正是两种适配器实现方式命名的由来。

由于两种实现方式只有ClassAdapter这个类是不同的，所以这里只给出此类的代码。

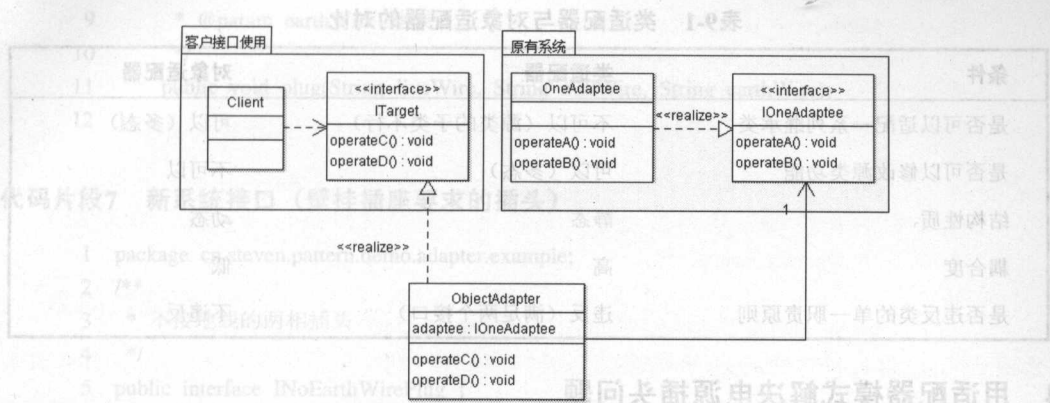


图9-3 对象适配器模型图

代码片段5 ObjectAdapter对象适配器

```
1 package cn.steven.pattern.demo.adapter;
2 /**
3  * 对象适配器
4  */
5 public class ObjectAdapter implements ITarget {
6     //委托对象
7     private IOneAdaptee adaptee;
8     //构造方法，传入一个源对象
9     public ObjectAdapter(IONeAdaptee oneAdaptee) {
10         this.adaptee = oneAdaptee;
11     }
12     //目标接口中的方法
13     public void operateC() {
14         //在此可使用委托对象的各种方法
15     }
16     //目标接口中的方法
17     public void operateD() {
18         //在此可使用委托对象的各种方法
19     }
20 }
```

由代码片段5可以领会到对象适配器的编程要点。对象适配器类中含有一个源类的接口（如果源类没有接口，直接用类声明也可以），在创建适配器时要设置此属性。这样，在适配器中的方法里就可以使用此对象了。

9.2.3 两种适配器的对比

学习了上述两种适配器的实现方式，可以看到，大部分情况下，两种适配器都可以满足转换接口的需求，但是它们之间还是有一定差别的，如表9-1所示。

由此可见，在编程中通常倾向于使用对象适配器从而实现程序更大的灵活性。这个结果也体现了面向对象的一个设计原则“优先使用（对象）组合，而非（类）继承”。

表9-1 类适配器与对象适配器的对比

条件	类适配器	对象适配器
是否可以适配一系列继承类	不可以（源类的子类不行）	可以（多态）
是否可以修改源类功能	可以（多态）	不可以
结构性质	静态	动态
耦合度	高	低
是否违反类的单一职责原则	违反（满足两个接口）	不违反

9.2.4 用适配器模式解决电源插头问题

在前面提出了家电区的麻烦的一节中已经描述过现在存在的电源插座的问题，学过了上述两种适配器，下面我们就可以解决这个问题了。在这里，给出的是对象适配器的实现方式，读者可以自行做出对应的类适配器实现方式。

解决方案如图9-4所示。

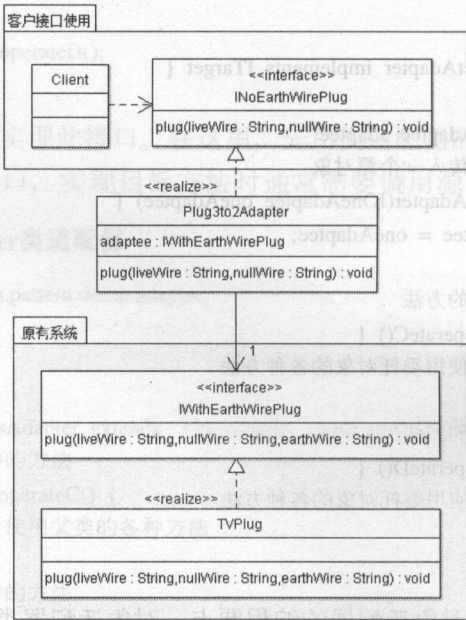


图9-4 插座问题的解决方案

先给出原有系统和客户系统的两种接口：

代码片段6 原有系统接口（电器自带的插头）

```
1 package cn.steven.pattern.demo.adapter.example;
2 /**
3  * 带接地线的插头
4  */
5 public interface IWithEarthWirePlug {
6     /**
7      * @param liveWire 火线
8      * @param nullWire 零线
```

```

9      * @param earthWire 地线
10     */
11     public void plug(String liveWire, String nullWire, String earthWire);
12 }

```

代码片段7 新系统接口（壁挂插座要求的插头）

```

1 package cn.steven.pattern.demo.adapter.example;
2 /**
3  * 不接地线的两相插头
4  */
5 public interface INoEarthWirePlug {
6     /**
7      * @param liveWire 火线
8      * @param nullWire 零线
9      */
10     public void plug(String liveWire, String nullWire);
11 }

```

由以上两个接口可以看出，方法的定义是不同的，也就是说客户端无法直接使用原有接口。下面看到的是源类和适配器类：

代码片段8 源类（实现了源接口）

```

1 package cn.steven.pattern.demo.adapter.example;
2 /**
3  * 电视机的带接地插头源类
4  */
5 public class TVPlug implements IWithEarthWirePlug {
6     public void plug(String liveWire, String nullWire, String earthWire){
7         //接通电源就可以工作了
8         play();
9     }
10    //工作方法
11    private void play(){
12        System.out.println("播放电视节目");
13    }
14 }

```

代码片段9 适配器类（对象适配器）

```

1 package cn.steven.pattern.demo.adapter.example;
2
3 /**
4  * 电源插头3转2适配器
5  */
6 public class Plug3to2Adapter implements INoEarthWirePlug {
7     // 源接口
8     private IWithEarthWirePlug adaptee;
9
10    // 实现

```



```

12     public void plug(String liveWire, String nullWire) {
13         // 接上两根线，地线悬空即可
14         adaptee.plug("火线", "零线", null);
15     }
16
17     /**
18      * 构造方法
19      * @param adaptee 传入一个适配源
20      */
21     public Plug3to2Adapter(IWithEarthWirePlug adaptee) {
22         this.adaptee = adaptee;
23     }
24
25 }

```

注意代码片段9的第9行和第21行~第23行是对源对象的声明和初始化。在这里采用了构造时传入源对象。在声明adaptee属性时使用了接口，可以适配同一类的源对象，比如假设还有一种电冰箱是三相插头的，也需要做转换，也可以使用此适配器。面向接口编程使程序有了更大的灵活性。

代码片段9的第14行展示了适配的方法，其实并没有做很复杂的操作，只是在委托源接口的时候给了一个空参数，这样就满足了源接口。

下面看一下客户端程序：

代码片段10 适配器客户端代码

```

1 package cn.steven.pattern.demo.adapter.example;
2 /**
3  * 客户端
4  */
5 public class Client {
6     public static void main(String[] args) {
7         //客户端接口，使用适配器对源类进行转换
8         INoEarthWirePlug plug = new Plug3to2Adapter(new TVPlug());
9         //插上插头
10        plug.plug("火线", "零线");
11    }
12 }

```

客户端运行结果如图9-5所示。

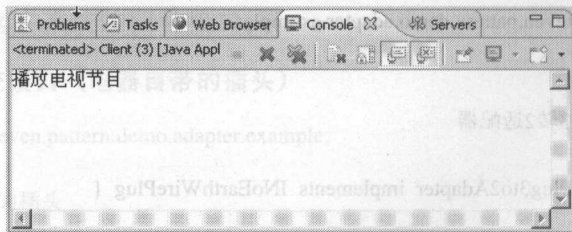


图9-5 客户端运行结果

至此，我们的工作终于完成了，带地线的三相插头终于插在了两相插座上。

思考:

- c) 如果有其他三相插头电器也需要进行这种转换, 是否需要重新定义适配器类?
- d) 如果电器不只插头不能用, 电压也不符合, 需要怎样写适配器类?
- e) 是否有一种适配器可以把很多种插头形式转换成两相插头?
- f) 是否有一种适配器可以把任何一种插头转换成任何形式的另一种插头?

9.2.5 双向适配器

双向适配器是同时实现了源接口和目标接口的适配器。在客户端方可以把双向适配器当做转换后的目标接口使用, 而在源系统方可以把双向适配器当做实现了源接口的类使用。按照这种思路, 可以生成三向适配器乃至多向适配器来适应多个系统, 最简单的方法是使适配器成为这些系统的子类, 当然, 在不支持多重继承 (如Java C#) 的语言中, 实现是比较复杂的。

在上述类适配器和对象适配器中, 类适配器天生就是双向适配器 (类适配器也是源类继承体系中的类, 自然实现了源的接口), 而对象适配器却只实现了目标接口, 不是双向适配器, 其改造方法如图9-6所示。

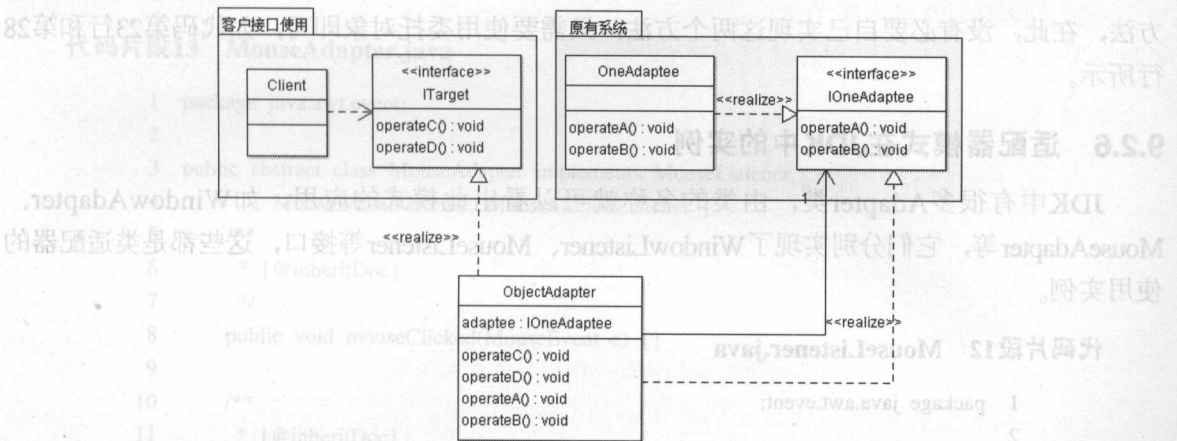


图9-6 双向对象适配器模型图

代码片段11 ObjectAdapter双向适配器

```
1 package cn.steven.pattern.demo.adapter;
2 /**
3  * 双向对象适配器
4  */
5 public class ObjectAdapter implements ITarget, IOneAdaptee {
6     //委托对象
7     private IOneAdaptee adaptee;
8     //构造方法, 传入一个源对象
9     public ObjectAdapter(OneAdaptee oneAdaptee) {
10         this.adaptee = oneAdaptee;
11     }
12     //目标接口中的方法
13     public void operateC() {
```

```

14      //在此可使用委托对象的各种方法
15    }
16    //目标接口中的方法
17    public void operateD() {
18        //在此可使用委托对象的各种方法
19    }
20    //源接口中的方法
21    public void operateA() {
22        //直接委托实现
23        adaptee.operateA();
24    }
25    //源接口中的方法
26    public void operateB() {
27        //直接委托实现
28        adaptee.operateB();
29    }
30 }

```

由图9-6和代码片段11可以学习到双向适配器的编程要点，即编写的适配器类必须同时实现目标接口和源接口，所以这里在对象适配器的基础上多实现了一个接口，代码中多写了两个方法，在此，没有必要自己实现这两个方法，只需要使用委托对象即可，如代码第23行和第28行所示。

9.2.6 适配器模式在JDK中的实例

JDK中有很多Adapter类，由类的名称就可以看出此模式的应用，如WindowAdapter、MouseListener等，它们分别实现了WindowListener、MouseListener等接口，这些都是类适配器的使用实例。

代码片段12 MouseListener.java

```

1  package java.awt.event;
2
3  import java.util.EventListener;
4
5  public interface MouseListener extends EventListener {
6
7      /**
8       * Invoked when the mouse button has been clicked (pressed
9       * and released) on a component.
10     */
11     public void mouseClicked(MouseEvent e);
12
13     /**
14      * Invoked when a mouse button has been pressed on a component.
15      */
16     public void mousePressed(MouseEvent e);
17
18     /**
19      * Invoked when a mouse button has been released on a component.
20      */

```



```

21 public void mouseReleased(MouseEvent e);
22
23 /**
24  * Invoked when the mouse enters a component.
25  */
26 public void mouseEntered(MouseEvent e);
27
28 /**
29  * Invoked when the mouse exits a component.
30  */
31 public void mouseExited(MouseEvent e);
32 }

```

由代码片段12可见, 应用接口中充斥着大量的方法, 编程者如果要实现这些接口必须实现所有的方法, 由于这些接口非常常用, 而实际运用时我们往往只关心这些接口的某几个方法, 但实现接口要求必须实现接口的所有方法, 以至于编程者经常不得不将接口方法实现为空。这实在是一件非常烦琐的事情, 并且, 代码中充斥着一些空方法, 也会影响阅读, 为此, JDK通过随接口一同发布的Adapter类为这些接口方法提供了默认的空方法, 例如代码片段13展示了其中一种适配器类。

代码片段13 MouseAdapter.java

```

1 package java.awt.event;
2
3 public abstract class MouseAdapter implements MouseListener,
4         MouseWheelListener, MouseMotionListener {
5     /**
6      * {@inheritDoc}
7      */
8     public void mouseClicked(MouseEvent e) {}
9
10    /**
11     * {@inheritDoc}
12     */
13    public void mousePressed(MouseEvent e) {}
14
15    /**
16     * {@inheritDoc}
17     */
18    public void mouseReleased(MouseEvent e) {}
19
20    /**
21     * {@inheritDoc}
22     */
23    public void mouseEntered(MouseEvent e) {}
24
25    /**
26     * {@inheritDoc}
27     */
28    public void mouseExited(MouseEvent e) {}
29

```

```

30  /**
31   * {@inheritDoc}
32   * @since 1.6
33   */
34  public void mouseWheelMoved(MouseWheelEvent e){}
35
36  /**
37   * {@inheritDoc}
38   * @since 1.6
39   */
40  public void mouseDragged(MouseEvent e){}
41
42  /**
43   * {@inheritDoc}
44   * @since 1.6
45   */
46  public void mouseMoved(MouseEvent e){}
47  }

```

由代码片段13可见，此抽象类实现了各种鼠标监听器接口，编程实现这些接口时如果直接实现接口，必须实现所有的方法，而继承这个类的话，只需要实现需要的方法即可，如现在需要做一个监听器类打印出鼠标移动时所在的坐标，则编码可以如代码片段14所示。

代码片段14 MyMouseListener.java

```

1  class MyMouseListener extends MouseAdapter {
2      @Override
3      public void mouseMoved(MouseEvent e) {
4          System.out.println("当前鼠标坐标为 X/Y:"
5              + e.getX() + "/" + e.getY());
6      }
7  }

```

当然，在实际应用中，更常见的是使用匿名类，编码如代码片段15所示。

代码片段15 MouseAdapter匿名类

```

1  button.addMouseListener(new MouseAdapter() {
2      @Override
3      public void mouseMoved(MouseEvent e) {
4          System.out.println("当前鼠标坐标为 X/Y:"
5              + e.getX() + "/" + e.getY());
6      }
7  });

```

由此可见，我们只需要从Adapter类派生（实际上，由于Adapter类往往被用于编写匿名内部类，因此，连派生这一步都省了，编译器会自动为我们完成这一步），并为需要处理的接口方法提供实现即可，从而节省了大量的编程工作量，这种做法通常也被称为**默认适配器**。

另一种情况就是传统用途的适配器，在Java 1.0中，遍历集合使用的是枚举接口，如代码片段16所示。

代码片段16 Enumeration.java

```

1 package java.util;
2
3 public interface Enumeration<E> {
4     /**
5      * Tests if this enumeration contains more elements.
6      *
7      * @return <code>true</code> if and only if this enumeration object
8      *         contains at least one more element to provide;
9      *         <code>false</code> otherwise.
10     */
11     boolean hasMoreElements();
12
13     /**
14      * Returns the next element of this enumeration if this enumeration
15      * object has at least one more element to provide.
16      *
17      * @return the next element of this enumeration.
18      * @exception NoSuchElementException if no more elements exist.
19     */
20     E nextElement();
21 }

```

Java 1.2以后的版本，修改了集合框架，引入了Iterator接口来遍历集合，如代码片段17所示。

代码片段17 Iterator.java

```

1 package java.util;
2
3 public interface Iterator<E> {
4     /**
5      * Returns <code>true</code> if the iteration has more elements.
6      * (In other words, returns <code>true</code> if <code>next</code>
7      * would return an element
8      * rather than throwing an exception.)
9      *
10     * @return <code>true</code> if the iterator has more elements.
11     */
12     boolean hasNext();
13
14     /**
15      * Returns the next element in the iteration.
16      *
17      * @return the next element in the iteration.
18      * @exception NoSuchElementException
19      *         iteration has no more elements.
20     */
21     E next();
22
23     /**
24      *
25      * Removes from the underlying collection the last element

```



```

26 * returned by the iterator (optional operation). This
27 * method can be called only once per call to <tt>next</tt>.
28 * The behavior of an iterator is unspecified if the
29 * underlying collection is modified while the iteration
30 * is in progress in
31 * any way other than by calling this method.
32 *
33 * @exception UnsupportedOperationException
34 * if the <tt>remove</tt> operation is not
35 * supported by this
36 * Iterator.
37 *
38 * @exception IllegalStateException
39 * if the <tt>next</tt> method has not yet
40 * been called, or the <tt>remove</tt> method
41 * has already been called after
42 * the last call to the <tt>next</tt> method.
43 */
44 void remove();
45 }

```

由于在Java 1.2版本后，普遍使用Iterator接口来遍历集合Collection，但是由Java 1.2版本之前的标准写成的代码使用的是Enumeration接口，这就造成了系统接口不满足用户接口的情况。

如下代码使用适配器模式进行了Iterator接口到Enumeration接口的转换。

代码片段18 Collections.java部分代码

```

1 /**
2  * Returns an enumeration over the specified collection. This provides
3  * interoperability with legacy APIs that require an enumeration
4  * as input.
5  *
6  * @param c the collection for which an enumeration is to be returned.
7  * @return an enumeration over the specified collection.
8  * @see Enumeration
9  */
10 public static <T> Enumeration<T> enumeration(final Collection<T> c) {
11     return new Enumeration<T>() {
12         Iterator<T> i = c.iterator();
13
14         public boolean hasMoreElements() {
15             return i.hasNext();
16         }
17
18         public T nextElement() {
19             return i.next();
20         }
21     };
22 }

```

由代码片段18的部分代码可见，通过这个静态方法，就可以把一个具有Iterator接口的Collection实例转换成Enumeration接口来使用，于是使用Enumeration来编程的旧式Java代码也可以

使用新式的集合类了。这正体现了适配器模式的意图: Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

9.2.7 适配器的适用范围

在以下情况下可以使用适配器模式:

- 系统需要使用现有的类, 而此类的接口不符合系统的要求。
- 想要建立一个可以重复使用的类, 用于与一些彼此之间没有太大关联的一些类, 包括一些可能在将来引进的类一起工作。这些源类不一定有很复杂的接口。
- (对对象适配器而言) 在设计里, 需要改变多个已有子类的接口, 如果使用类的适配器模式, 就要针对每一个子类做一个适配器, 而这不太实际。

9.2.8 与其他模式的关系

许多初学者认为适配器、代理和装饰都是通过组合一个现存对象, 通过调用该方法来实现自己的功能的, 它们之间很相像, 其实结构型模式都是以继承和组合的方式来实现新的功能, 如果单看实现过程, 它们的确很相像, 但如果从意图上分析, 它们的区别就大了。

代理模式着重将复杂部分抽到中间层, 通过这个中间层(代理层)来控制对目标对象的访问, 它要求代理层和目标对象的接口相同。而适配器模式解决的恰恰是接口发生了变化导致现有对象不能工作的问题, 它通过组合这个现有对象, 将现有接口转化为目标接口。

装饰模式强调的是通过组合来动态扩展对象的功能。比如前面所述的收音机, 本身具有播放功能。现在有很多种录音机(英语录音机、中文录音机), 可在收音机上组装一种录音机, 使其除了具有播放功能外还具有录音功能, 而且组装不同的录音机, 将具有不同语言的录音功能。这种需求就是装饰模式的应用场景。它和适配器模式在意图上有明显的不同。

以上三种模式中装饰模式和适配器模式都有一个别名叫包装模式, 但包装的形式是不一样的, 下面着重对比一下这两种模式。

定义上:

• 装饰模式: 以对客户端透明的方式扩展对象的功能, 是继承关系的一个替代方案, 提供比继承更多的灵活性。使用原来被装饰的类的一个子类的实例, 把客户端的调用委派到被装饰类。

• 适配器模式: 把一个类的接口变换成客户端所期待的另一种接口, 从而使原本因接口不匹配而无法一起工作的两个类能够一起工作。适配类可以根据参数返还一个合适的实例给客户端。

从定义上看装饰模式是对核心对象或者功能的扩展, 适配器模式是把对象或者功能放到一个新对象中引用。举个例子, 现在书店卖的关于道德经的书, 有线装版, 有精装版; 有日文版, 有英文版, 其中线装版和精装版就是装饰模式, 日文版和英文版就是适配器模式, 各种版本都是为迎合不同消费者的不同需求。为什么这么说呢? 因为线装版和精装版的道德经虽然包装不同, 但内容相同。日文版和英文版就不同, 这两个版本的内容就可能和原版的不同, 翻译的内容虽来自道德经, 但根据不同国家的文化, 思维逻辑的不同就可能改变一些行文方式。

使用条件:

• 装饰模式一般在下列情况使用: 需要扩展一个类的功能或者给一个类增加附加责任; 需要动态地给一个对象增加功能, 这些功能可以再动态撤销; 需要增加有一些基本功能的排列组

合而产生非常多的功能，从而使得继承关系变得不现实。

- 适配器模式一般用于：系统需要使用现有的类，但此类已经不符合系统的要求；想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。适配器模式在系统升级的时候使用的频率很高，可使旧系统的一些功能方法在新系统中引用。

Java中的应用：

装饰模式和适配器模式在Java中的I/O文件的操作中都有体现。

Java的IO库中处理流的类有FileInputStream, FileOutputStream, DataInputStream, DataOutputStream类等。在InputStream, OutputStream, Reader, Writer结构的内部，有一些流处理器可以对另一些流处理器起到装饰作用，形成新的、改善的流处理器。这就体现了装饰模式的作用。同时在一些流处理器的内部有对其他流处理器的功能的适配引用，这体现了适配器模式的优点。

9.3 适配器模式总结

Adapter模式允许我们使用现有类进行扩展来满足客户类的需求。当现有的类不能满足客户需要的接口时，通常可以创建一个新类来实现接口和扩展现有类。这种方法会创建一个类适配器，它将把客户的调用转变为调用现有类的方法。除此之外，还可以使用现有类的实例来创建一个新的客户子类。这种方法会创建一个对象适配器，将客户调用转发给现有类的实例。这两种方法分别为类适配器和对象适配器。

当客户需要在两个以上的抽象层次中都使用适配器时，通常会让适配器类实现两个以上的接口，它称为双向适配器。

当设计系统需要使用到遗留系统时，要考虑从使用Adapter模式的软件架构中所带来的好处和灵活性。

第10章 桥接模式 (Bridge)

桥接模式将抽象部分与实现部分分离，使它们都可以独立变化¹。

在日常生活中，经常会遇到这样一类问题：产品由很多部分组成，在描述工作时就需要按照不同部分的组成对其进行细分，但不能使用同一种构建过程生产，同时也不能由一个功能点方便地调用另一个功能点的作用。

如需要一部交通工具，可以先抽象出交通工具的几个功能部分，如轮子、座位数量、排挡方式等，根据实际需要针对每一个功能点指定特定的实现，最后就可以得到这部交通工具了。比如需要一部8轮、15座、自动挡的车，再如需要一部4轮、5座、手动挡的车，使不同功能的抽象与其实现相分离，这就是桥接模式方便我们完成的这种需求的架构设计。

10.1 游戏机销售专柜引发的思考

在超市的游戏机销售专柜，经理发现了这样一个问题，有各种不同厂家生产的游戏机，也有不同厂家生产的游戏手柄，这些设备需要按照客户的意愿自由进行组合（这里只是假设），如何才能让不同的设备组合在一起呢？

首先，来看一下在这个需求中有什么可以确定的抽象：

- 游戏主机：PS3、XBOX360、WII……
- 游戏手柄：4键8方向手柄、手枪手柄、方向盘手柄……

然后考虑如何让这些设备组装在一起，经过思考，设计出的架构如图10-1所示。

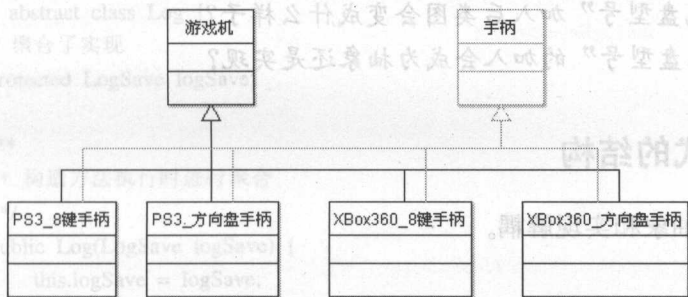


图10-1 游戏机配件的初步设计

由图10-1可以看出这种设计是极为复杂的，况且此图并没有列出所有的组合情况，经过分析，如上设计至少有以下几个缺点：

- 抽象的不断增多导致组合类的数量膨胀。
- 抽象部分多样化导致修改一个类的某一种抽象时容易破坏其他抽象需求。
- 增加和删除抽象十分困难。

图10-1中游戏机和手柄抽象类的设计是没有问题的，但是生成的子类却同时维护了两个抽

¹Decouple an abstraction from its implementation so that the two can vary independently. GOF[95]

象级别的变化，如果现在要增加一种“游戏光盘型号”抽象，那么设计好的所有实现类都需要修改，这导致了结构的不灵活，原因就是抽象与实现耦合在一起了。

在这里，最容易迷惑的就是什么是抽象，什么是实现，在此例中，抽象就是“游戏机”，而一种实现就是“手柄”。

在此处，必须了解两个概念才能明白桥接模式的意义：

• 解耦（Decouple）：是指让各种事物互相独立地行事，或者至少明确地声明它们之间的关系。

• 抽象（Abstraction）：是指不同事物之间概念上的联系方式。

桥接模式所要做的就是对抽象和实现部分进行解耦，如果在设计中产生了过度的耦合，这通常是由于使用了过度的继承导致的。其原因是面向对象分析师很喜欢使用继承来解决碰到的各种问题。继承的处理方法看上去新颖且功能强大，但是许多面相对象设计的关注点都放在了通过派生处理变化，也就是通过派生类来实现新功能上。

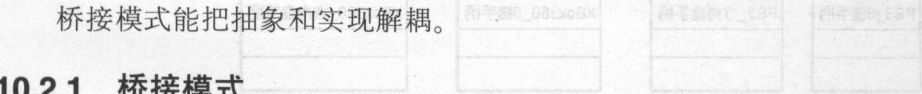
从总体上看，面向对象是简单的，因为对象可以分层次，整个系统的功能可以被描述为几个高层对象的交互。从具体对象看，每个对象有一个“是什么”的明确概念，有一组可以“做什么”的外部特征，好的面向对象设计中，这两者是一致的，这也是“谨慎继承”原则在面向对象设计中被强调的原因。在面向对象设计方法中，有两种重用对象的方式，一种是继承，另一种是聚合。继承方式的重用实现起来非常简单，以至于它容易被滥用。在使用继承方式之前，需要谨慎考虑“IS-NESS”的问题，即新的对象是不是那个希望重用的对象。

由此可见，在游戏机销售问题中是不应该使用继承来使一个类同时实现两种抽象的。下一节中，我们将学习使用桥接模式解决此问题。

思考：

- a) 为什么不说“游戏机”是实现，而“手柄”是抽象？
- b) “游戏光盘型号”加入后类图会变成什么样子？
- c) “游戏光盘型号”的加入会成为抽象还是实现？

10.2 桥接模式的结构



桥接模式能把抽象和实现解耦。

10.2.1 桥接模式

为了说明桥接模式的使用方法，下面使用一个例子进行讲解。

假设设计一个日志系统，这个系统可以记录多种日志类型，如交易日志、数据库日志、用户操作日志等，同时，这个系统还支持多种日志的表现方式，如XML文件、文本文件、数据库数据、E-mail等。

很明显，在日志系统中有两种独立的抽象变化：日志的种类和日志的表现方式。如果一个系统有两个以上相互独立的抽象变化，就需要使用桥接模式了。

桥接模式是把抽象和实现相分离，先来看一下在日志系统中什么是抽象，什么是实现。

- 抽象：日志的种类。
- 实现：日志的表现方式。

它们各自都有自己的变化，根据模式定义，设计出的类图如图10-2所示。

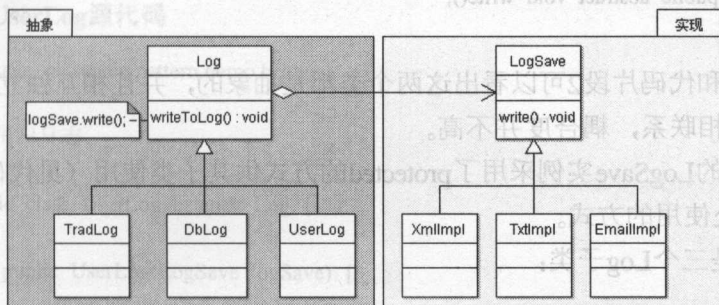


图10-2 桥接模式UML类图

由图10-2可见，采用桥接模式的设计，现在抽象和实现已经能独立地变化而不相互影响了。在此基础上，如果要新增一种独立变化的实现，也不会对原有结构造成影响，这就符合了“开一闭原则”。

下面来重点分析一下实现代码，首先看一下两个高层的抽象类Log和LogSave:

代码片段1 Log源代码

```
1 package cn.steven.pattern.demo.bridge;
2
3 /**
4  * 日志抽象类
5  */
6 public abstract class Log {
7     // 聚合了实现
8     protected LogSave logSave;
9
10    /**
11     * 构造方法执行时进行聚合
12     */
13    public Log(LogSave logSave) {
14        this.logSave = logSave;
15    }
16
17    //此处抽象由具体实现类的实现
18    public abstract void writeToLog();
19 }
```

代码片段2 LogSave源代码

```
1 package cn.steven.pattern.demo.bridge;
2
3 /**
4  * 日志保存抽象类
5  */
```



```
6 public abstract class LogSave {
7
8     /**
9      * 具体保存方法由实现类实现
10    */
11    public abstract void write();
12 }
```

由代码片段1和代码片段2可以看出这两个类都是抽象的，并且相互独立，Log类采用聚合的方式与LogSave相联系，耦合度并不高。

Log类中聚合的LogSave实例采用了protected的方式供其子类使用（见代码片段1第8行），读者需要注意此处使用的方式。

其次展示的是三个Log子类：

代码片段3 TradLog源代码

```
1 package cn.steven.pattern.demo.bridge;
2 /**
3  * 交易日志
4  */
5 public class TradLog extends Log {
6
7     public TradLog(LogSave logSave) {
8         super(logSave);
9     }
10
11     @Override
12     public void writeToLog() {
13         System.out.println("写入TradLog数据");
14         //调用桥接过来的对象
15         this.logSave.write();
16     }
17
18 }
```

代码片段4 DbLog源代码

```
1 package cn.steven.pattern.demo.bridge;
2 /**
3  * 数据库日志
4  */
5 public class DbLog extends Log {
6
7     public DbLog(LogSave logSave) {
8         super(logSave);
9     }
10
11     @Override
12     public void writeToLog() {
13         System.out.println("写入DbLog数据");
14         //调用桥接过来的对象
```

```
15         this.logSave.write();
16     }
17
18 }
```

代码片段5 UserLog源代码

```
1 package cn.steven.pattern.demo.bridge;
2 /**
3  * 用户日志
4  */
5 public class UserLog extends Log {
6
7     public UserLog(LogSave logSave) {
8         super(logSave);
9     }
10
11     @Override
12     public void writeToLog() {
13         System.out.println("写入UserLog数据");
14         //调用桥接过来的对象
15         this.logSave.write();
16     }
17
18 }
```

由代码片段3、代码片段4和代码片段5可见在设计抽象部分的子类时，子类并不知道使用的是哪一种具体实现（在这里，就是不知道使用的是哪一种日志表现方式），只需要调用父类聚合好的使用抽象描述的对象即可。

最后展示的是LogSave的实现类：

代码片段6 XmlImpl源代码

```
1 package cn.steven.pattern.demo.bridge;
2 /**
3  * xml日志存储
4  */
5 public class XmlImpl extends LogSave {
6
7     @Override
8     public void write() {
9         System.out.println("使用xml方式存储");
10     }
11
12 }
```

代码片段7 TextImpl源代码

```
1 package cn.steven.pattern.demo.bridge;
2 /**
3  * 文本日志存储
```

```
4  */
5  public class TextImpl extends LogSave {
6
7      @Override
8      public void write() {
9          System.out.println("使用文本方式存储");
10     }
11
12 }
```

代码片段8 EmailImpl源代码

```
1  package cn.steven.pattern.demo.bridge;
2  /**
3   * 邮件日志存储
4   */
5  public class EmailImpl extends LogSave {
6
7      @Override
8      public void write() {
9          System.out.println("使用发邮件的方式存储");
10     }
11
12 }
```

代码片段6、代码片段7和代码片段8展示了LogSave类的三种独立变化的实现。

注意，实现不能主动和抽象发生耦合，而只能由抽象和实现发生耦合，在本例中，就是要由Log主动和LogSave发生耦合。

下面由启动类展示最终的结果：

代码片段9 Client源代码

```
1  package cn.steven.pattern.demo.bridge;
2
3  public class Client {
4      public static void main(String[] args) {
5          //在构建对象时使用桥接模式
6          Log log = new UserLog(new XmlImpl());
7          log.writeToLog();
8
9          //在构建对象时使用桥接模式
10         Log log2 = new DbLog(new EmailImpl());
11         log2.writeToLog();
12     }
13
14 }
```

代码片段9的运行结果，如图10-3所示。

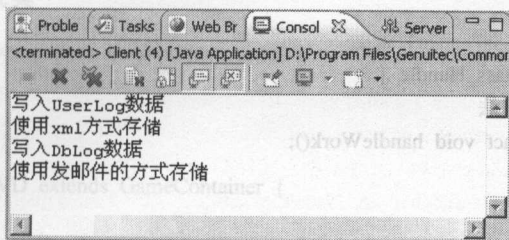


图10-3 代码片段9运行结果

至此，我们已经完成了使用桥接模式进行日志系统设计的案例，在下一节中，将进一步运用桥接模式进行更复杂问题的设计。

10.2.2 使用桥接模式解决游戏机销售的问题

经过上一节的学习，现在就可以来解决游戏机问题了，游戏机销售所面临的问题是客户可以自由选择游戏主机、手柄、游戏光盘等配件，在将配件组装起来后，各部件之间可以互相调用，并且可以根据需要更换部件，如换一个手柄。

在这个问题中，可以发现存在不同的抽象体系，并且这些抽象可以相互独立地变化。在更换一种抽象的实现后，不影响其他抽象的功能。

使用桥接模式的结构，可以设计类图如图10-4所示。

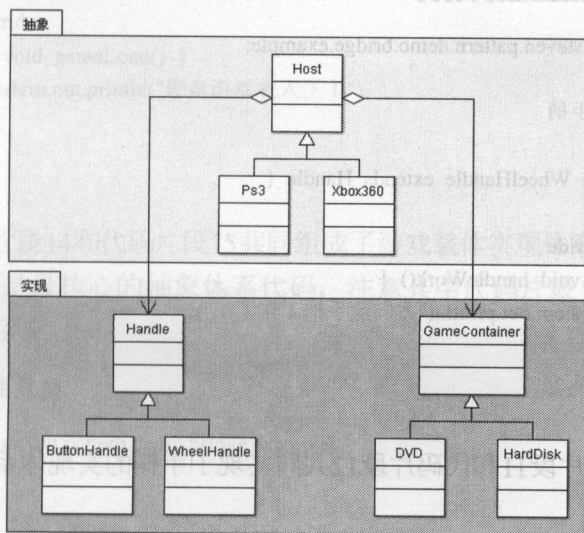


图10-4 游戏机销售问题的桥接模式结构设计

由图10-4可见，抽象与实现已经分离了，并且一个抽象如果有实现上的变化不会影响其他抽象。抽象是稳定的，实现是可变的。所以在编程中要面向抽象编程，这样实现类如果要修改或更换，应用代码就无需改变。

下面先展示的是手柄实现，包括其抽象类和实现类：

代码片段10 Handle抽象类

```
1 package cn.steven.pattern.demo.bridge.example;
2 /**
```

```

3  * 手柄抽象类
4  */
5  public abstract class Handle {
6      //手柄工作方法
7      public abstract void handleWork();
8  }

```

代码片段11 ButtonHandle实现类

```

1  package cn.steven.pattern.demo.bridge.example;
2  /**
3   * 按钮手柄
4   */
5  public class ButtonHandle extends Handle {
6
7      @Override
8      public void handleWork() {
9          System.out.println("按钮手柄工作了!");
10     }
11
12 }

```

代码片段12 WheelHandle实现类

```

1  package cn.steven.pattern.demo.bridge.example;
2  /**
3   * 方向盘手柄
4   */
5  public class WheelHandle extends Handle {
6
7      @Override
8      public void handleWork() {
9          System.out.println("方向盘手柄工作了!");
10     }
11
12 }

```

代码片段10、代码片段11和代码片段12共同实现了手柄的实现体系，这个体系可以独立于其他体系的变化。

下面展示的是游戏载体实现。

代码片段13 GameController抽象类

```

1  package cn.steven.pattern.demo.bridge.example;
2
3  /**
4   * 游戏载体抽象类
5   */
6  public abstract class GameController {
7      // 游戏载体工作方法
8      public abstract void gameLoad();
9  }

```

代码片段14 DVD实现类

```
1 package cn.steven.pattern.demo.bridge.example;
2 /**
3  * DVD游戏
4  */
5 public class DVD extends GameContainer {
6
7     @Override
8     public void gameLoad() {
9         System.out.println("DVD游戏载入了!");
10    }
11
12 }
```

代码片段15 HardDisk实现类

```
1 package cn.steven.pattern.demo.bridge.example;
2
3 /**
4  * 硬盘游戏
5  */
6 public class HardDisk extends GameContainer {
7
8     @Override
9     public void gameLoad() {
10        System.out.println("硬盘游戏载入了!");
11    }
12
13 }
```

代码片段13、代码片段14和代码片段15共同组成了游戏载体实现体系，也可以独立地变化。

下面展示的代码就是最核心的抽象体系代码，注意其中代码片段16 Host抽象类中是怎样聚合了另外两个实现体系的。

代码片段16 Host抽象类

```
1 package cn.steven.pattern.demo.bridge.example;
2
3 /**
4  * Host抽象类
5  */
6 public abstract class Host {
7     /**
8      * 注意此处聚合的两个保护属性是
9      * 桥接模式的精华所在，实现了抽
10     * 象和实现的分离
11     */
12
13     protected Handle handle;
14
15     protected GameContainer gameContainer;
```



```

16
17 /**
18  * 构造方法
19  * @param handle Handle的一个具体实现对象
20  * @param gameContainer gameContainer的一个具体实现对象
21  */
22 public Host(Handle handle, GameContainer gameContainer) {
23     super();
24     this.handle = handle;
25     this.gameContainer = gameContainer;
26 }
27
28 // 游戏机的工作方法
29 public abstract void work();
30
31 /**
32  * 注意此处的两个配件工作方法,
33  * 把功能委托给了聚合的属性
34  */
35
36 public void handleWork() {
37     handle.handleWork();
38 }
39
40 public void gameLoad() {
41     gameContainer.gameLoad();
42 }
43 }

```

注意代码片段16中第13行、第15行为聚合属性；第36行~第42行为委托方法，供其子类使用，这样子类就无需直接调用聚合的对象了，同时还对子类隐藏了聚合对象工作方法的实现细节。

下面展示的是两个游戏主机的实现类。

代码片段17 Ps3实现类

```

1 package cn.steven.pattern.demo.bridge.example;
2
3 /**
4  * Ps3游戏主机
5  */
6 public class Ps3 extends Host {
7
8     public Ps3(Handle handle, GameContainer gameContainer) {
9         super(handle, gameContainer);
10    }
11
12    @Override
13    public void work() {
14        System.out.println("Ps3工作了！");
15        // 调用委托方法
16        this.gameLoad();
17        this.handleWork();

```

```
18 }
19 package cn.steven.pattern.demo.bridge.example;
20 }
```

代码片段18 Xbox360实现类

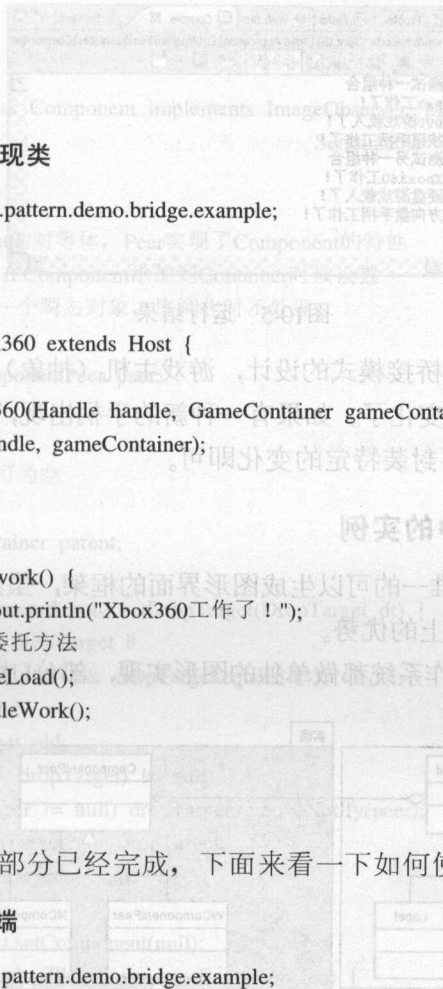
```
1 package cn.steven.pattern.demo.bridge.example;
2 /**
3  * Xbox360主机
4  */
5 public class Xbox360 extends Host {
6
7     public Xbox360(Handle handle, GameContainer gameContainer) {
8         super(handle, gameContainer);
9     }
10
11     @Override
12     public void work() {
13         System.out.println("Xbox360工作了!");
14         // 调用委托方法
15         this.gameLoad();
16         this.handleWork();
17     }
18
19 }
```

至此，桥接模式的架构部分已经完成，下面来看一下如何使之运行。

代码片段19 Client客户端

```
1 package cn.steven.pattern.demo.bridge.example;
2
3 /**
4  * 可以更换配件的游戏机测试类
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9         System.out.println("测试一种组合");
10        Host host1 = new Ps3(new ButtonHandle(), new DVD());
11        host1.work();
12
13        System.out.println("测试另一种组合");
14        Host host2 = new Xbox360(new WheelHandle(), new HardDisk());
15        host2.work();
16    }
17
18 }
```

运行结果如图10-5所示。



运行结果如图10-5所示。

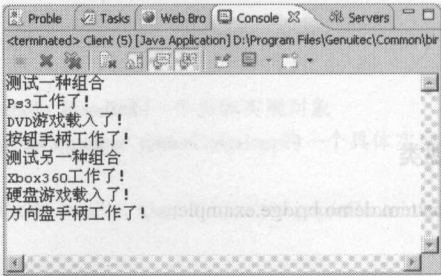


图10-5 运行结果

由运行结果可见，经过了桥接模式的设计，游戏主机（抽象）与各种配件（实现）终于可以互不影响、而是相互独立地变化了。如果有一种新的手柄出现，就不需要增加一个耦合了抽象及其他实现的类了，只需要封装特定的变化即可。

10.2.3 桥接模式在JDK中的实例

Java刚刚推出时，Awt是唯一的可以生成图形界面的框架，虽然现在Swing后来居上，不过Awt依然在本地代码上有速度上的优势。

Awt的原理是对每一种操作系统都做单独的图形实现，部分U桥接模式类图如图10-6所示。

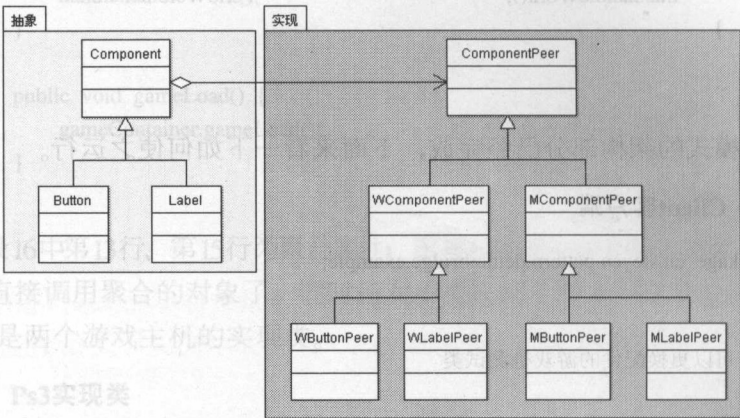


图10-6 JDK中部分U桥接模式类图

如图10-6所示，JDK中采用桥接的方式解决了在不同操作系统上显示各种图形组件的问题。在这里的抽象是Component及其子类，图形组件有很多种，并且可以独立变化，如按钮、标签、文本框、滚动条等。它们在不同的图形系统平台上显示的方式都不一样，所以实现必须和抽象相分离，因而有了Component的对等体ComponentPeer及其实现类。WComponentPeer和MComponentPeer是不同的图形平台，WButtonPeer、WLabelPeer、MButtonPeer、MLabelPeer是不同图形平台上针对各种Component的实现。

代码片段20 Component代码节选

```
1 /*
2  * @(#)Component.java 1.444 08/12/12
3  *
4  * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
5  * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
```



```

6  */
7  package java.awt;
8
9  public abstract class Component implements ImageObserver, MenuContainer,
10     Serializable
11  {
12     /**
13      * Component的对等体, Peer实现了Component的特性
14      * Peer 将会在Component添加到Container时被设置
15      * 注: 这是一个瞬态对象, 序列化时不处理
16      */
17     transient ComponentPeer peer;
18
19     /**
20      * 父容器, 可为空
21      */
22     transient Container parent;
23
24     public synchronized void setDropTarget(DropTarget dt) {
25         if (dt == dropTarget ||
26             (dropTarget != null && dropTarget.equals(dt)))
27             return;
28         DropTarget old;
29         if ((old = dropTarget) != null) {
30             if (peer != null) dropTarget.removeNotify(peer);
31             DropTarget t = dropTarget;
32             dropTarget = null;
33             try {
34                 t.setComponent(null);
35             } catch (IllegalArgumentException iae) {
36                 // ignore it.
37             }
38         }
39     }
40 }

```

代码片段20展示了如何使抽象桥接实现。在其第17行使用瞬态描述的原因是：不同的图形平台显示图形的方式都不一样，所以在序列化传输这个对象时与平台相关的代码可以不传输，在反序列化时再生成特定平台的对等对象。

代码片段21 Button代码节选

```

1  /*
2   * @(#)Button.java    1.83 08/10/31
3   *
4   * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
5   * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
6   */
7
8  package java.awt;
9
10 public class Button extends Component implements Accessible {

```

```

11     public void addNotify() {
12         synchronized(getTreeLock()) {
13             if (peer == null)
14                 peer = getToolkit().createButton(this);
15             super.addNotify();
16         }
17     }
18 }

```

由代码片段21可见抽象的子类中有大量的代码使用到了抽象类中聚合的具体实现。这样具体的抽象实现类就无需关心特定图形平台的实现方法了。

ComponentPeer接口定义了大量的图形操作方法。

代码片段22 ComponentPeer代码节选

```

1  /*
2   * @(#)ComponentPeer.java    1.52 08/01/23
3   *
4   * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
5   * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
6   */
7
8  package java.awt.peer;
9
10 public interface ComponentPeer {
11     void setVisible(boolean b);
12     void setEnabled(boolean b);
13     void paint(Graphics g);
14     void print(Graphics g);
15     void handleEvent(AWTEvent e);
16     void coalescePaintEvent(PaintEvent e);
17     Point getLocationOnScreen();
18     Dimension getPreferredSize();
19     Dimension getMinimumSize();
20     ColorModel getColorModel();
21     Toolkit getToolkit();
22     Graphics getGraphics();
23     FontMetrics getFontMetrics(Font font);
24     void dispose();
25     void setForeground(Color c);
26     void setBackground(Color c);
27     void setFont(Font f);
28     void updateCursorImmediately();
29 }

```

WComponentPeer.java类定义了Windows中操作图形的基本方法，WButtonPeer.java定义了Windows中画出Button控件的方法。这两个类的代码由于篇幅原因就不列出了，具体代码请参见OpenJDK。

由JDK中Awt框架的例子可见，当需要抽象与实现相分离时，桥接模式是很好的解决方案。

10.2.4 桥接模式的使用范围

在以下情况下应当使用桥接模式：

- 如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系时。
- 设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。
- 一个构件有多于一个的抽象化角色和实现化角色，系统需要在它们之间进行动态耦合时。
- 虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，实际情况下需要独立管理这两者时。

效果及实现要点：

- **Bridge**模式使用“对象间的组合关系”解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。
- 所谓抽象和实现沿着各自维度的变化，即“子类化”它们，在得到各个子类之后，便可以获得不同的组合对象。
- **Bridge**模式有时候类似于多继承方案，但是多继承方案往往违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差。**Bridge**模式是比多继承方案更好的解决方法。
- **Bridge**模式一般应用在“两个非常强的变化维度”时。有时候即使有两个变化的维度，但是某个方向的变化维度并不剧烈——两个变化不会导致纵横交错的结果，便不一定要使用**Bridge**模式。

10.2.5 与其他模式的关系

桥接模式和装饰模式：

这两个模式在一定程度上都可减少子类的数目，避免出现复杂的继承关系，但是它们解决的方法却各有不同，装饰模式把子类中比基类多出来的部分放到单独的类里面，以适应新功能的增加，把描述新功能的类封装到基类的对象里面时，就得到了所需要的子类对象，这些描述新功能的类通过组合可以实现很多的功能组合。而桥接模式是把两个以上独立的抽象维度分离，使用聚合的方式使其关联，来达到减少子类的目的，结构上桥接模式比装饰模式要复杂。

桥接模式和适配器模式：

它们的共同点是桥接和适配器都是让两个类配合工作，它们的区别是出发点不同，适配器的出发点是改变已有的两个接口，让它们相容。桥接模式的出发点是分离抽象化和实现化，使两者的接口可以不同，目的是分离。

10.3 桥接模式总结

桥接模式是一个非常有用的模式，也非常复杂，它很好地符合了开-闭原则，并且优先使用对象，而不是继承这两个对象。

当需求中相互关联的抽象有多个变化维度时，需要使用桥接模式来解耦。

第11章 代理模式 (Proxy)

代理模式为其他对象提供一种代理以控制对这个对象的访问¹。

这个模式的用意就是使用一个对象来替代另一个对象。在现实生活中，也经常遇到“代理”组成的词，比如有一家电脑公司，代理销售联想品牌的电脑，这个代理商本身是不生产电脑的，而且甚至可以没有电脑，但是可以通过这家公司买到联想的电脑，就好像直接向联想厂商购买一样。再比如去银行存钱，如果柜台的人过多，通常会去ATM机取钱，ATM机也可以操作银行系统，但是在它上面能做的事情却没有柜台的多。

可见，由于某种原因无法与想要操作的对象直接交互的话，可以通过一个中介对象来间接实现访问的目的。有很多种原因会需要采用间接访问，比如时间不允许等待，在春运季节通过代理点买火车票就是这种情况的一个实例；再如安全原因，老板的电话不想亲自接而让秘书接也是一个例子，如果想给外国友人寄送一个物品，但是由于时间和路途问题无法直接送给对方，于是我们通过委托快递公司的方式间接送给友人，也是一样的道理。

代理模式的作用就是通过一个中间层达到间接访问目标对象的目的。

11.1 手机柜台遇到的问题

到手机柜台选购手机的人川流不息，这个柜台的手机样式众多，品牌很多，销售人员忙个不停，有的在帮客户讲解各款手机的性能，有的在帮助顾客试用手机。

销售人员的工作非常繁忙，但是他们有一个重要的任务：当客户在试用手机时要注意是否有人拿走手机。

此类问题的一个通常解决方案是职责单一化，比如客户选定商品后需要到另一个房间试手机，在另一房间里有相应的人员来管理安全问题，但是这样一来客户的选购过程就变得麻烦了，有可能导致客户不再选购手机，这样超市就得不偿失了，超市必须解决这样一个问题，就是既让顾客可以方便地挑选和试用手机，又让销售人员在顾客试用时可以不看管顾客，提高工作效率。

另一个问题就是超市卖的手机是由各个厂家生产的，销售人员众多，而且销售柜台也很多，销售人员在卖一款手机的时候很难知道厂家是否有货，而且由于距离和时间的问题也不可能马上拿到指定的手机商品。

如何解决上述的问题呢？如果不能或不方便直接访问对象的话，可以通过代理模式来达到访问的目的。

¹Provide a surrogate or placeholder for another object to control access to it. GOF[95]

11.2 代理模式的结构

代理模式使客户代码通过间接的方式访问目标代码。

11.2.1 代理模式

在软件中，如果两个软件模块或两个抽象层次不能直接互相访问，则可以通过增加一个层次来解决这个问题。下图为代理模式的简易软件架构图，如图11-1所示。

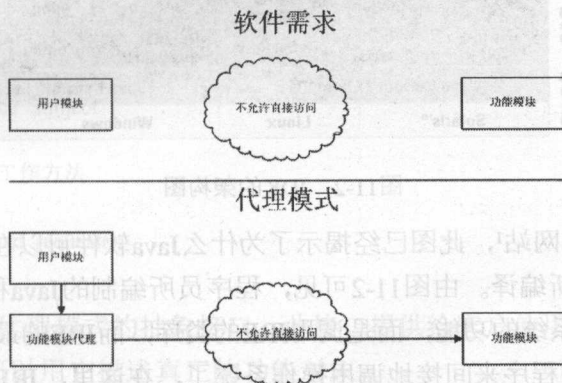


图11-1 代理模式的简易软件架构示意图

由图11-1可见，如果两个软件模块的需求是一方调用另一方的话，由于某种原因不能实现，可以在本地模块增加一个代理模块来间接调用对方模块。

其实在软件体系中有很多设计都可以用代理模式的原理去理解。比如操作系统中的系统软件，传统定义为“计算机软件分为系统软件和应用软件两大类，系统软件用于管理计算机本身和应用程序，应用软件是为满足用户特定需求而设计的软件，操作系统是最基本的系统软件，它和系统工具软件构成了系统软件”。操作系统的功能可归纳为CPU管理、存储器管理、设备管理、文件管理。但由于CPU管理很复杂，可分为静态管理和动态管理，所以一般将中央处理器管理又分为作业管理和进程管理两个部分。定义中只是说明了操作系统是系统软件，将其引申则其意图会更加明显：操作系统就是代理访问计算机硬件系统的一个抽象层次，各种应用软件事会通过调用操作系统的功能间接操作硬件。

现在来思考一下：为什么不让应用软件直接操作硬件？

世界上存在很多种硬件体系，如IBM兼容机、MAC机、Sun公司的Solaris，这些硬件的操作指令都很复杂，不但要用大量的时间去学习操作硬件的方法，而且这样设计的话，各种软件的安全是没有办法控制的，各种黑客软件可以直接控制硬件，所以基于易用性和安全性的原因就需要操作系统做中间层。以上只是设计操作系统中间层的一部分原因，已足以说明设计代理的重要性了。

下面再来思考一下什么Java程序可以跨平台运行？

要解释这个问题，先来看一下JDK的架构图，如图11-2所示。

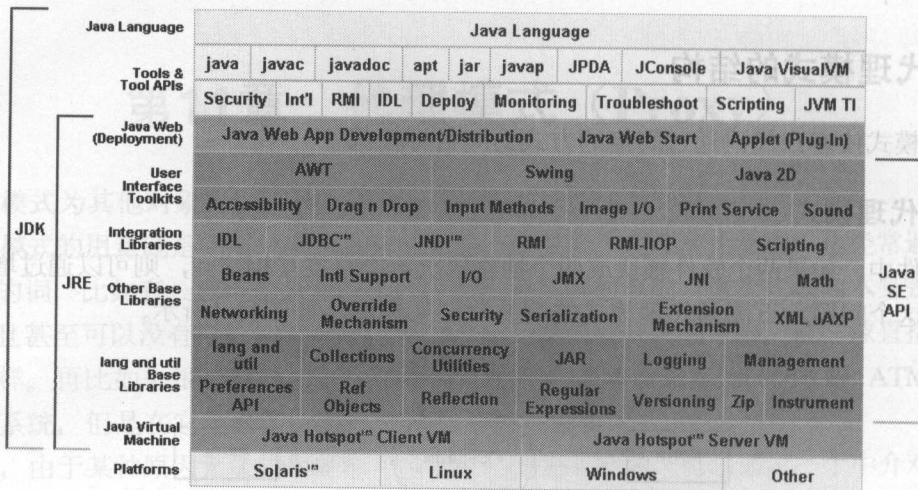


图11-2 JDK的架构图

图11-2来自Java官方网站¹，此图已经揭示了为什么Java软件可以在多种操作系统上运行而不用做修改，也不用重新编译。由图11-2可见，程序员所编制的Java程序都是运行于JRE之上的，程序不会调用操作系统的功能，而是调用JRE的类库，而JRE的最低层是各种操作系统，于是JRE就代替程序员的程序来间接地调用操作系统了，在这里，JRE也起到了代理的作用，它的好处不言而喻，程序员可以不用理会各种操作系统的不同之处进行编程，编出来的Java程序就自然可以跨平台了。注意，编写优秀的跨平台Java程序还需要注意一些问题，请参见实现Java程序跨平台运行的注意事项²。

GOF所设计的代理模式结构图，如图11-3所示。

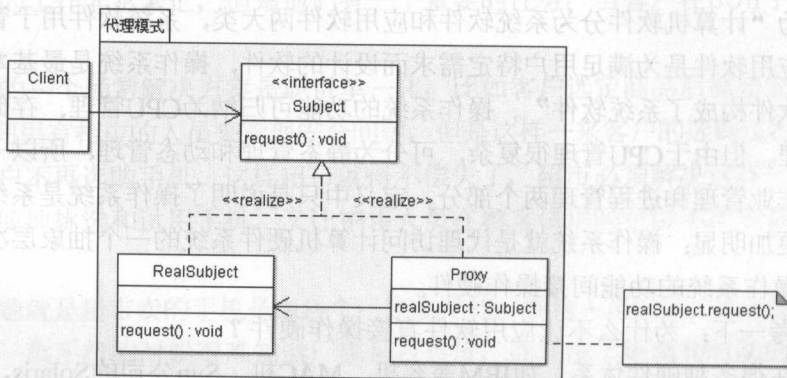


图11-3 代理模式结构图

- **Subject接口:** 定义了`RealSubject`和`Proxy`的共用接口，这样在任何使用`RealSubject`的地方都可以使用`Proxy`。
- **RealSubject:** 定义`Proxy`可以代表的实体。
- **Proxy:** 保存一个引用使代理可以访问`RealSubject`实体。通常和`RealSubject`一样也实现了

¹<http://java.sun.com/javase/7/docs/>。

²<http://blog.csdn.net/shadowkiss/archive/2009/10/13/4662492.aspx>。

Subject接口，它控制着对RealSubject实体的访问，并能负责创建和销毁它。

由图11-3可以看到代理模式的结构相对比较简单，它的特点就是不直接访问真正的对象而使用代理对象间接调用对象。

下面来看一下组成代理模式的各部分代码。

代码片段1 Subject.java

```
1 package cn.steven.pattern.demo.proxy;
2
3 /**
4  * 代理模式的主题接口
5  * 此接口供客户使用，用于操作代理类对象
6  */
7 public interface Subject {
8     /**
9      * 抽象工作方法
10     */
11     public void request();
12 }
```

代码片段1展示的是代理模式的抽象接口，此接口提供给客户使用，用于描述代理实例，在代理实例的代码中还可以用它描述真正的功能对象。

代码片段2 RealSubject.java

```
1 package cn.steven.pattern.demo.proxy;
2
3 /**
4  * 主题实现类
5  * 提供客户需要的功能
6  */
7 public class RealSubject implements Subject {
8
9     /**
10     * 构造方法
11     */
12     public RealSubject() {
13     }
14
15     /**
16     * 实现业务方法
17     */
18     @Override
19     public void request() {
20         System.out.println("运行业务功能");
21     }
22
23 }
```

代码片段2展示的是真正的业务对象，在此软件体系中，为真正的业务功能实现者，但是由于某种原因，客户端无法直接调用它。

代码片段3 Proxy.java

```
1 package cn.steven.pattern.demo.proxy;
2
3 /**
4  * 代理类
5  */
6 public class Proxy implements Subject {
7
8     private Subject realSubject;
9
10    /**
11     * 构造方法
12     */
13    public Proxy() {
14    }
15
16    // 代理业务功能
17    @Override
18    public void request() {
19        /**
20         * 采用延时懒加载的方式
21         * 把对象的实例化拖延至使用时创建
22         */
23        if (realSubject == null) {
24            realSubject = new RealSubject();
25        }
26        System.out.println("代理调用业务方法开始: ");
27        realSubject.request();
28        System.out.println("代理调用业务方法完毕!");
29    }
30
31 }
```

代码片段3展示的是代理类的代码，注意代码第8行使用接口来描述真正的业务实体，第23行~第25行使用懒加载（Lazy loading¹）的方式进行业务实体的实例化，用以改善系统的性能。

下面来看一下客户调用的方法：

代码片段4 Client.java

```
1 package cn.steven.pattern.demo.proxy;
2
3 /**
4  * 代理模式客户端
5  */
6 public class Client {
7     public static void main(String[] args) {
8         // 使用接口描述代理实例
9     }
10 }
```

¹http://en.wikipedia.org/wiki/Lazy_load.

```

9      Subject subject = new Proxy();
10     subject.request();
11 }
12 }

```

代码片段4的第9行~第10行为客户代码，由此可见客户代码没有直接调用业务对象，而是采用调用代理对象的方式来间接调用业务对象，这在某些时候是必需的办法。

运行结果如图11-4所示。

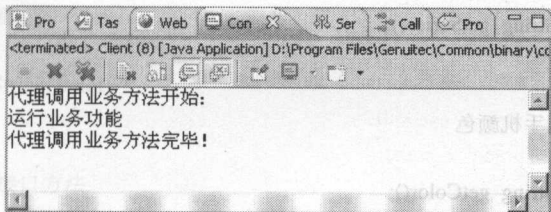


图11-4 运行结果

由图11-4可见客户通过代理对象采用懒加载的方式调用了真正的业务对象的业务方法，在代理对象调用业务方法时，还可以在方法内加入需要处理的其他业务逻辑，比如安全验证、日志处理、加密处理、编码处理等以满足客户的不同需求。

11.2.2 使用代理模式解决手机展示过程中的问题

经过了上一节的学习，下面就可以应用代理模式来解决手机柜台上的销售问题。

为了使客户在挑选手机时更加方便，而且销售人员也不用担心安全问题，可以设计一种手机模型，它具有真实手机的外表材料、按键、重量等特点，但是不具备通话的功能，这种模型的成本很低廉，所以不用过分担心丢失。

客户在挑选的过程中就可以使用这种手机模型进行参考。由功能可见，手机模型有很多功能和真机是一样的，但是不具备所有的功能，我们可以把它看做是真机的一个代理，这个代理的作用是限制业务对象的功能。客户选定了手机款式后，再要求销售人员拿出真机购买。

销售人员和销售柜台有很多，各款手机的流动量也很大，所以需要有一个业务对象来统一管理手机，在这里，设计的是一个仓库类，这个类根据实际情况被设计为单例的。仓库类的作用是为销售人员提供手机货源，而且可根据需要向各大手机商采购进货。设计的类图如图11-5所示。

注意图11-5中的各个类的方法和属性并没有全部列出，注意设计中的DummyPhone类充当了MobilePhone的代理，它的大部分方法都是委托MobilePhone的实例实现的，但是由于DummyPhone不能拨打电话，所以call方法不能委托实现。

首先展示的是接口的代码：

代码片段5 Phone.java

```

1 package cn.steven.pattern.demo.proxy.example;
2
3 /**
4  * 手机接口
5  */

```



```
6 public interface Phone {
7     /**
8      * 得到手机品牌
9      */
10    public String getBrand();
11
12    /**
13     * 得到手机型号
14     */
15    public String getSeries();
16
17    /**
18     * 得到手机颜色
19     */
20    public String getColor();
21
22    /**
23     * 接通电话
24     */
25    public void call();
26 }
```

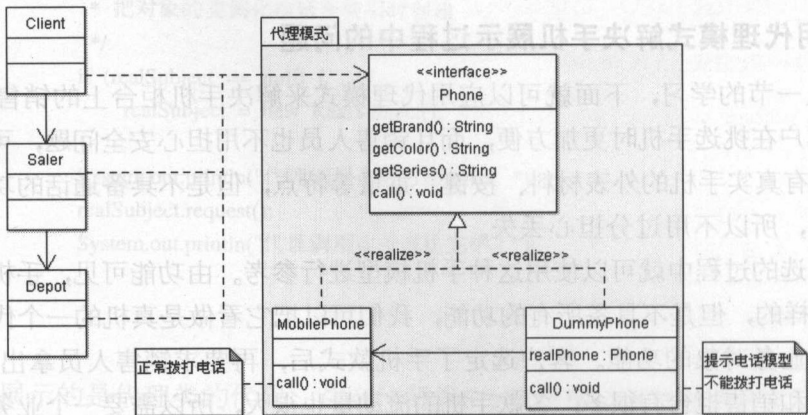


图11-5 代理模式解决手机销售问题UML类图

代码片段5展示的是代理模式的接口，参照UML类图可见它将被多个类使用，注意接口在软件结构中是耦合度最低的设计。

下面展示的是具体的业务类，也就是被代理的类（真实的手机）的代码：

代码片段6 MobilePhone.java

```
1 package cn.steven.pattern.demo.proxy.example;
2
3 /**
4  * 手机实现类，此类不允许直接使用
5  */
6 public class MobilePhone implements Phone {
7
8     /**
9      * 品牌名称
```

```

10      */
11      private String brand;
12
13      /**
14       * 手机型号
15       */
16      private String series;
17
18      /**
19       * 手机颜色
20       */
21      private String color;
22
23      /**
24       * 重写接口方法
25       */
26      @Override
27      public String getBrand() {
28          return brand;
29      }
30
31      public void setBrand(String brand) {
32          this.brand = brand;
33      }
34
35      /**
36       * 重写接口方法
37       */
38      @Override
39      public String getSeries() {
40          return series;
41      }
42
43      public void setSeries(String series) {
44          this.series = series;
45      }
46
47      /**
48       * 重写接口方法
49       */
50      @Override
51      public String getColor() {
52          return color;
53      }
54
55      public void setColor(String color) {
56          this.color = color;
57      }
58
59      /**
60       * 重写接口方法

```

```

61 public interface Phone {
62     @Override
63     public void call() {
64         System.out.println(getColor() + "颜色的" +
65             getBrand() + getSeries()
66             + "手机开始拨打电话!");
67     }
68     /**
69      * 带参数的构造方法
70      */
71     public MobilePhone(String brand, String series,
72         String color) {
73         this.brand = brand;
74         this.series = series;
75         this.color = color;
76     }
77     /**
78      * 不带参数构造方法
79      */
80     public MobilePhone() {
81     }
82     /**
83      * 用于Map中比较是否一致
84      * 需要重写equals方法
85      */
86     @Override
87     public boolean equals(Object obj) {
88         MobilePhone mp = null;
89         try {
90             mp = (MobilePhone) obj;
91         } catch (Exception e) {
92             // 如果转换出现异常则返回假
93             return false;
94         }
95         if (getBrand().equals(mp.getBrand())
96             && getColor().equals(mp.getColor())
97             && getSeries().equals(mp.getSeries())) {
98             return true;
99         } else {
100             return false;
101         }
102     }
103     /**
104      * 用于Map中比较是否一致
105      * 需要重写hashCode方法

```



```

112 * 按照Java的要求, equals和
113 * hashCode必须结果一致
114 * 这里的算法只起到演示作用
115 */
116 @Override
117 public int hashCode() {
118     return getBrand().hashCode() & getColor().hashCode()
119         & getSeries().hashCode();
120 }
121 }

```

代码片段6是真实手机的实现代码, 注意其中第85行~第120行是为进行比较而重写Object的两个方法。这么做的原因可以参看JDK的文档, 摘录如下:

public boolean equals(Object obj)

指示其他某个对象是否与此对象“相等”。

equals方法在非空对象引用上实现相等关系:

- 自反性: 对于任何非空引用值x, x.equals(x)都应返回true。
- 对称性: 对于任何非空引用值x和y, 当且仅当y.equals(x)返回true时, x.equals(y)才应返回true。
- 传递性: 对于任何非空引用值x、y和z, 如果x.equals(y)返回true, 并且y.equals(z)返回true, 那么x.equals(z)应返回true。
- 一致性: 对于任何非空引用值x和y, 多次调用x.equals(y)始终返回true或始终返回false, 但前提是对象上进行equals比较所用的信息没有被修改。
- 对于任何非空引用值x, x.equals(null)都应返回false。

对于任何非空引用值x和y, 当且仅当x和y引用同一个对象时, Object类的equals方法才返回true (x == y具有值true)。

注意: 当此方法被重写时, 通常有必要重写hashCode方法, 以维护hashCode方法的常规协定, 该协定声明相等对象必须具有相等的哈希码。

参数: obj-要与之比较的引用对象。

返回: 如果此对象与obj参数相同, 则返回true; 否则返回false。

另请参见: hashCode(), Hashtable。

public int hashCode()

返回该对象的哈希码值。使用此方法是为了提高哈希表(例如java.util.Hashtable提供的哈希表)的性能。

hashCode的常规协定是:

- 在Java应用程序的执行期间, 在对同一对象多次调用hashCode方法时, 必须一致地返回相同的整数, 前提是将对象进行equals比较时所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行, 该整数无需保持一致。
- 如果根据equals(Object)方法, 两个对象是相等的, 那么对这两个对象中的每个对象调用hashCode方法都必须生成相同的整数结果。
- 如果根据equals(java.lang.Object)方法, 两个对象不相等, 那么对这两个对象中的任

一对象调用hashCode方法不要求一定生成不同的整数结果。但是，程序员应该意识到，为不相等的对象生成不同整数结果可以提高哈希表的性能。

实际上，由Object类定义的hashCode方法确实会针对不同的对象返回不同的整数。（这一般是通过将该对象的内部地址转换成一个整数来实现的，但是JavaTM编程语言不需要这种实现技巧。）

返回：此对象的一个哈希码值。

另请参见：equals(java.lang.Object), Hashtable。

由文档可见，如果想判断两个对象是否相等，不能只使用equals方法，还必须使两个对象的hashCode值相等才具有完备性。代码片段6第118行使用的是一种简易的方法，此算法容易算出同样的值来，如果想做出很完善的类，需要仔细斟酌hashCode的算法。

下面来看一下如何设计手机模型代理：

代码片段7 DummyPhone.java

```

1  package cn.steven.pattern.demo.proxy.example;
2
3  /**
4   * 手机模型
5   */
6  public class DummyPhone implements Phone {
7
8      /**
9       * 需要代理的业务对象
10      */
11      private Phone realPhone;
12
13      public Phone getRealPhone() {
14          return realPhone;
15      }
16
17      public void setRealPhone(Phone realPhone) {
18          this.realPhone = realPhone;
19      }
20
21      /**
22       * 带参数的构造方法
23       */
24      public DummyPhone(String brand, String series,
25                          String color) {
26          realPhone = new MobilePhone(brand, series, color);
27      }
28
29      /**
30       * 不带参数的构造方法
31       */
32      public DummyPhone() {
33
34

```

```

35
36 /**
37  * 重写接口方法
38  */
39 @Override
40 public void call() {
41     System.out.println(getColor() + "颜色的" +
42         getBrand() + getSeries()
43         + "手机模型无法拨打电话!");
44 }
45
46 /**
47  * 重写接口方法
48  */
49 @Override
50 public String getBrand() {
51     return realPhone.getBrand();
52 }
53
54 /**
55  * 重写接口方法
56  */
57 @Override
58 public String getColor() {
59     return realPhone.getColor();
60 }
61
62 /**
63  * 重写接口方法
64  */
65 @Override
66 public String getSeries() {
67     return realPhone.getSeries();
68 }
69
70 }

```

代码片段7中大部分的接口方法都委托给了真实的手机实例，但是第36行~第44行使用了不能工作的方法，体现了手机模型的功能局限性。

下面是销售人员的代码：

代码片段8 Saler.java

```

1 package cn.steven.pattern.demo.proxy.example;
2
3 /**
4  * 销售人员
5  */
6 public class Saler {
7
8     /**
9      * 挑选手机，销售人员提供模型机

```



```

10    */
11    public Phone choosePhone(String brand,
12        String series, String color) {
13        return new DummyPhone(brand, series, color);
14    }
15
16    /**
17     * 购买手机，此时销售人员提供真机
18     */
19    public Phone buyPhone(String brand, String series,
20        String color) {
21        Phone phone = Depot.getInstance().getPhone(
22            brand, series, color);
23        if (phone == null) {
24            System.out.println(color + "色" + brand +
25                series + "手机无货");
26            return null;
27        }
28        System.out.println("购买" + phone.getColor() +
29            "色" + phone.getBrand()
30            + phone.getSeries() + "一部");
31        return phone;
32    }
33
34 }

```

代码片段8中只设计了提供模型手机和真机的方法供客户调用，注意在提供真机时是调用了仓库类的相关方法。

下面展示手机仓库的代码：

代码片段9 Depot.java

```

1  package cn.steven.pattern.demo.proxy.example;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  /**
7   * 具有采购和储存功能的仓库
8   */
9  public class Depot {
10     /**
11      * 单例模式使用的自身实例
12      */
13     private static Depot depot;
14
15     /**
16      * 仓库当前储备
17      */
18     private static Map<Phone, Integer> reserve;
19
20     /**

```

```
21 * 单例模式获得仓库实例
22 */
23 public static synchronized Depot getInstance() {
24     if (depot == null) {
25         depot = new Depot();
26     }
27     return depot;
28 }
29
30 /**
31  * 单例模式需要私有的构造方法
32  */
33 private Depot() {
34     // 构建储备库
35     reserve = new HashMap<Phone, Integer>();
36
37     // 放一些测试数据
38     reserve.put(new MobilePhone(
39         "NOKIA", "N95", "红"), 3);
40     reserve.put(new MobilePhone(
41         "APPLE", "iphone", "黑"), 5);
42 }
43
44 /**
45  * 在仓库中取出手机，此方法注意要同步
46  */
47 public synchronized static Phone getPhone(String brand,
48     String series,
49     String color) {
50     Phone phone = new MobilePhone(brand, series, color);
51
52     // 查看是否存在这款手机
53     if (reserve.containsKey(phone) &&
54         reserve.get(phone) > 0) {
55         // 库存减一
56         reserve.put(phone, reserve.get(phone) - 1);
57         return phone;
58     }
59     // 如果没有返回空
60     return null;
61 }
62
63 }
```

很明显，代码片段9是一个单例类，这也符合超市的实际情况，只有一个仓库。单例类的各种操作都需要注意多线程安全的问题，可以看到，代码中有很多方法都加上了 **synchronized** 关键字来保证多线程临界资源的安全。

最后看一下客户端的代码：

代码片段10 Client.java

```
1 package cn.steven.pattern.demo.proxy.example;
2
3 /**
4  * 顾客
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9         System.out.println("===测试手机模型===");
10        Saler saler = new Saler();
11        Phone phone = saler.choosePhone(
12            "NOKIA", "N95", "红");
13        phone.call();
14        System.out.println("===购买手机===");
15        phone = saler.buyPhone("APPLE", "iphone", "黑");
16        phone.call();
17        System.out.println("===购买无货手机===");
18        phone = saler.buyPhone("APPLE", "iphone", "绿");
19    }
20
21 }
```

代码运行结果如图11-6所示。

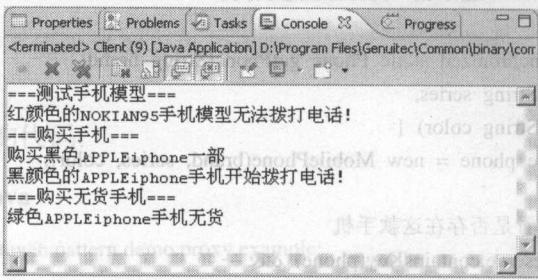


图11-6 代码运行结果

由图11-6可见，经过了上述的设计，顾客在挑选手机时拿到的是模型机，除了不能拨打电话外其他的功能和真机是一样的，真正需要购买时拿到的才是真机，所以销售人员节省了大量的工作精力来处理卖手机的过程。销售人员不需要管理货物的问题，每当顾客要买手机时直接向库房要货即可，这样也大幅提高了其工作的效率。

11.2.3 几种不同的代理模式

几种代理模式的构造方法都一样，但是用途不一样，按功能划分，大体分为以下八类：

- 远程代理（Remote）：为远程地址空间的对象创建一个本地的对象来代理其功能，常见于分布式应用、Web Service、远程过程调用等情况中。
- 虚拟代理（Virtual）：当要使用的对象需要大量的机器资源才能运行时，通常使用一个虚拟代理来间接调用它，把此对象延时到真正使用的时候才创建，使系统的效率提高。常见的是各种插件架构，所有的插件并不是加载系统时就会初始化，而是真正使用时才创建。

• **写时复制代理 (Copy-on-Write)**：虚拟代理的一种，开始对象都使用同一个值对象，在某个对象需要改变值的时候再把这个值对象复制在另一个地方进行修改。典型的应用如Java中的String对象的值不可改变，所以不同的String对象如果值相同则指向同一个内存地址，只有当值改变时才分配其他的内存空间。

• **保护代理 (Protect or Access)**：保护功能对象的调用，通常要进行权限方面的控制。常见的应用场景为对核心功能类设计保护代理使之访问的安全性得到保证。

• **缓存代理 (Cache)**：为某些对象进行缓存，使客户代码在调用这些对象时不必再进行运算或查询。常见于各种O/R Mapping框架中，如Hibernate JPA等。

• **防火墙代理 (FireWall)**：保护对象，使之不能被轻易访问。常见于开放式软件架构中。

• **同步化代理 (Synchronization)**：使多个对象访问同一个对象时不会产生多线程问题，如死锁、线程干扰等。常见于多线程程序中对临界资源的控制。

• **智能引用 (Smart Reference)**：当引用主题对象时，执行一些额外的操作，如统计引用计数，锁定对象，预备资源等工作。

在GOF的《设计模式》一书中，只提到了四种代理模式的应用，分别是远程代理、虚拟代理、保护代理和智能引用，因为这四种代理使用的频率很高。

在这里，为了说明现有的设计代码是怎样使用代理模式的，展示了几种现存的框架及类库的使用方法。

动态代理

从JDK1.3版本开始，Java就引入了动态代理的概念。动态代理 (Dynamic Proxy) 可以帮助你减少代码行数，真正提高代码的可复用度。例如，你不必为所有类的方法都写上相同的Log代码行，取而代之的是使用类的动态代理类。当然，这种便利是有条件的。

在JDK 1.3版本以前，代理模式就已经开始流行。代理模式是生成一个和类相同接口的代理类，用户通过使用代理类来封装某个实现类。其目的是加强实现类的某个方法的功能，而不必改变原有的源代码。代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

随着Proxy的流行，Sun公司把它纳入到JDK 1.3中实现了Java的动态代理。Java动态代理类位于Java.lang.reflect包下，一般主要涉及以下两个类。

• **Proxy**：提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。

• **InvocationHandler**：是代理实例的调用处理程序实现的接口。每个代理实例都具有一个关联的调用处理程序。对代理实例调用方法时，将对方法调用进行编码并将其指派到它的调用处理程序的invoke方法中。

使用这两个类的示例类图如图11-7所示。

图11-7中深色背景的为JDK提供的接口和类，其功能是使用Proxy类来动态生成Foo类的代理，客户端只需要使用IFoo接口即可。

下面展示的是具体实现代码：

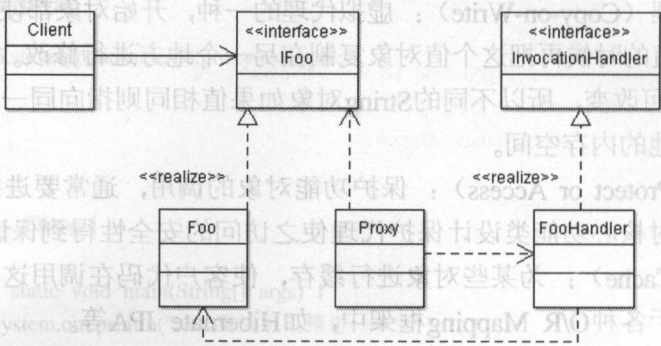


图11-7 JDK动态代理类图

代码片段11 IFoo.java

```
1 package cn.steven.pattern.demo.proxy.jdk;
2
3 /**
4  * 供动态代理使用的接口
5  */
6 public interface IFoo {
7
8     public String print();
9
10 }
```

下面是其实现类:

代码片段12 Foo.java

```
1 package cn.steven.pattern.demo.proxy.jdk;
2
3 /**
4  * 被代理的实现类
5  */
6 public class Foo implements IFoo {
7     /**
8      * 实现工作方法
9      */
10    public String print() {
11        System.out.println("Run Foo: print()");
12        return "返回值: value.";
13    }
14 }
```

实现代理:

代码片段13 FooProxy.java

```
1 package cn.steven.pattern.demo.proxy.jdk;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5 import java.lang.reflect.Proxy;
```

其运行结果如图11-8所示。

```

6
7 /**
8  * 动态代理
9  */
10 public class FooProxy {
11
12     /**
13      * 客户运行代码
14      */
15     public static void main(String[] args) {
16         // 生成动态代理对象
17         IFoo fooProxy = (IFoo) Proxy.newProxyInstance(
18             Thread.currentThread().getContextClassLoader(),
19             new Class[] { IFoo.class },
20             new FooHandler());
21         // 使用代理调用方法
22         System.out.println(fooProxy.print());
23     }
24
25 }
26
27 /**
28  * 创建代理类
29  */
30 class FooHandler implements InvocationHandler {
31
32     /**
33      * 实现调用方法
34      */
35     public Object invoke(Object proxy, Method method,
36         Object[] args)
37         throws Throwable {
38         /**
39          * 创建一个被代理的对象
40          */
41         Object obj = new Foo();
42         System.out.println(proxy.getClass().getName() +
43             " 将调用 " + obj.getClass().getName() +
44             " 的 " + method.getName() + " 方法!");
45         /**
46          * 调用被代理对象的方法
47          */
48         Object invoke = method.invoke(obj, args);
49         System.out.println("调用完毕!");
50         /**
51          * 返回被调用对象的返回值
52          */
53         return invoke;
54     }
55 }
56 }

```


其运行结果如图11-8所示。

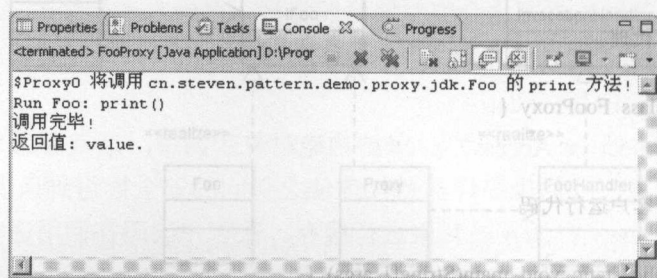


图11-8 动态代理运行结果

由图11-8可见，使用动态代理可以在不建立代理类的情况下就实现代理的效果，方便了编程工作的进行，在很多框架中都可以看到动态代理的使用。

Copy-On-Write代理

在Doug Lea的“Concurrent Programming in Java”一书中，对Copy-On-Write模式做了最好的描述。实际上，这个模式声明了，为了维护对象的一致性快照，要依靠不可变性（*immutability*）来消除在协调读取不同的但是相关的属性时需要的同步。对于集合，这意味着如果有大量的读（即`get()`）和迭代时，不必同步操作以照顾偶尔的写（即`add()`）调用。对于新的`CopyOnWriteArrayList`和`CopyOnWriteArraySet`类，所有可变的（*mutable*）操作都首先取得后台数组的副本，对副本进行更改，然后替换副本。这种做法保证了在遍历自身发生更改的集合时，永远不会抛出`ConcurrentModificationException`。遍历集合会用原来的集合完成，而在以后的操作中使用更新后的集合。

`CopyOnWriteArrayList`和`CopyOnWriteArraySet`这些新的集合，最适合用于读操作通常大大超过写操作的情况。一个最常提到的例子是使用监听器列表。前面已经说过，`Swing`组件还没有改为使用新的集合。相反，它们继续使用`javax.swing.event.EventListenerList`来维护它们的监听器列表。

由此可见，`Copy-On-Write`可以很有效率地在多线程的环境中保证数据和操作的安全性。在这里，以`CopyOnWriteArrayList`为例看一下如何实现此模式。

`CopyOnWriteArrayList`是`ArrayList`的一个线程安全的变体，其中所有可变操作（`add`、`set`等等）都是通过对底层数组进行一次新的复制来实现的。这一般需要很大的开销，但是当遍历操作的数量大大超过可变操作的数量时，这种方法可能比其他替代方法更有效。在不能或不想进行同步遍历，但又需要从并发线程中排除冲突时，它也很有用。“快照”风格的迭代器方法在创建迭代器时使用了对数组状态的引用。此数组在迭代器的生存期内不会更改，因此不可能发生冲突，并且迭代器保证不会抛出`ConcurrentModificationException`。创建迭代器以后，迭代器就不会反映列表的添加、移除或者更改。在迭代器上进行的元素更改操作（`remove`、`set`和`add`）不受支持，这些方法将抛出`UnsupportedOperationException`。

代码片段14 `CopyOnWriteArrayList.java`部分代码

```
1 public class CopyOnWriteArrayList<E> implements
2     List<E>, RandomAccess, Cloneable, java.io.Serializable {
3
4     private volatile transient Object[] array;
```

```

5
6  /**
7   * 增加元素时使用Copy-On-Write
8   */
9   public boolean add(E e) {
10      final ReentrantLock lock = this.lock;
11      lock.lock();
12      try {
13          Object[] elements = getArray();
14          int len = elements.length;
15          /**
16           * 注意此处代码需要把原array的值复制出一份再操作
17           */
18          Object[] newElements = Arrays.copyOf(elements, len + 1);
19          newElements[len] = e;
20          /**
21           * 把此集合指向的元素指向新的地址
22           */
23          setArray(newElements);
24          return true;
25      } finally {
26          lock.unlock();
27      }
28  }
29
30  /**
31   * 注意此处的迭代器访问的是当前array值
32   */
33  public Iterator<E> iterator() {
34      return new COWIterator<E>(getArray(), 0);
35  }
36  }

```

由代码片段14可见,在进行插入和删除等修改数据操作的情况下此类是使用复制-修改-修改原指针的方式处理安全问题的,每次使用iterator方法都指向的是当前数据地址,如果在迭代的过程中发生了修改,对迭代器所访问的数据是没有影响的,因为修改会在另一个内存空间进行,和读取互不干扰。

但是,此种解决方案只适合大量操作都是读取操作,少量操作是修改操作的情况,如果错用,有可能导致对性能的影响。

远程代理

Enterprise Java Bean (EJB) 是Sun Microsystems对CORBA的可移植性和复杂性提出的解决方案。EJB引入了比CORBA更简单的编程模块,它可以让开发人员创建可移植分布式组件,称做Enterprise Bean。EJB编程模块可以让开发人员创建安全的、事务性的和持久的商业对象(Enterprise Bean),该对象使用非常简单的编程模块和声明属性,与CORBA不同,例如访问控制(授权安全性)和事务管理等设施时它非常易于编程。CORBA需要使用复杂的API来利用这些服务,而EJB则根据一种称做“部署描述信息”的特性文件中的声明将这些服务自动应用到Enterprise Bean。这个模型确保了Bean开发人员可以集中精力编写商业逻辑,而容器会自动

管理更复杂但又必要的操作。

由于EJB规范颁布了一组明确的EJB容器（供应商服务器）和EJB组件（商业对象）之间的契约，因此EJB中实现了可移植性。这些契约或规则确切规定容器必须为Enterprise Bean提供什么服务，Bean开发人员需要使用什么API和声明属性来创建Enterprise Bean。由于详细指定了Enterprise Bean的生命周期，因此供应商知道如何在运行时管理Bean，Bean开发人员确切知道Enterprise Bean在其存在期间可以做什么。

Enterprise Java Bean简化了分布式对象的开发、部署和访问。EJB分布式对象（一种Enterprise Bean）的开发人员只需依照为Enterprise Java Bean建立的契约和协议实现对象。支持EJB的应用程序服务器可以使用任何分布式网络协议，包括本地Java RMI协议（JRMP）、专有协议或CORBA的网络协议（IIOP）。不管在某个特定产品中使用的基本网络协议是什么，EJB会使用相同的编程API和语义以Java RMI-IIOP访问分布式对象。协议的细节对应用程序和Bean开发人员隐藏；对于所有供应商来说，定位和使用分布式Bean的方法是相同的。

客户端在使用服务器端的EJB服务时，如何调用是一个关键性的问题，下图是EJB技术在分布式环境中的示意图，如图11-9所示。

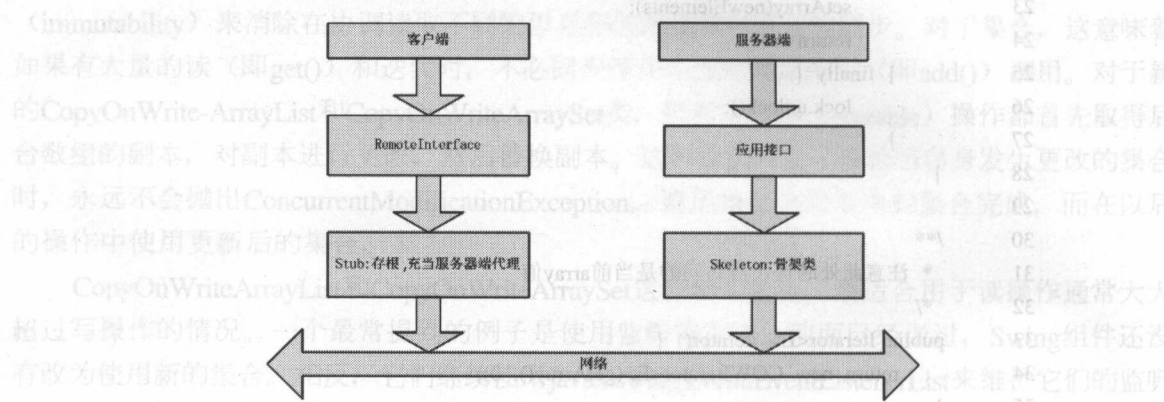


图11-9 EJB技术在分布式环境中的示意图

图11-9可见，在客户端的Stub就是一个远程代理，这样设计后，客户端就无需直接远程访问服务器端，只需要把Stub对象当做EJB服务调用即可，Stub处理了远程访问的问题，这是一种典型的远程代理的设计案例。

11.2.4 代理模式的使用范围

如果因为某种原因不想让或不能让客户代码可以直接调用业务实体的时候，需要使用代理模式。这时就需要编写代理类，代理类的功能、种类很多，下面使用举例的方式进行应用情况的说明。

- 一个对象，比如一幅很大的图像，需要载入的时间很长时。
- 一个需要很长时间才可以完成的计算结果，并且需要在它的计算过程中显示中间结果时。
- 一个存在于远程计算机上的对象，需要通过网络载入这个远程对象则需要很长时间，特别是在网络传输高峰期时。
- 一个对象只有有限的访问权限，代理模式（Proxy）可以验证用户的权限时。
- 一个对象需要被频繁地查询和引用时。

- 一个对象需要在多线程的使用中保证安全时。
- 一个对象被多个对象引用，当其中一个引用需要在被改写时被复制出来时。
- 一个对象使用时需要自动做其他的工作时。

11.2.5 与其他模式的关系

与代理模式非常相似的模式是装饰模式和适配器模式，先来描述一下这三者的区别：

- 代理模式是业务对象的一个替身，由于客户代码不能直接调用业务对象所以才通过代理对象来间接访问，通常由代理对象来维护业务对象的生命周期。
- 装饰模式是对业务对象的功能增强，客户要求的是功能增强，并不是不能使用业务对象，而且业务对象本身也是由客户代码创建的。
- 适配器模式是要改变业务对象的接口，使业务对象可以满足一个新的接口，它与代理模式的区别是代理模式的业务对象接口本身就是可用的，其目的是不一样的。

再来广义地比较一下代理模式与其他多种模式的区别，Proxy模式与Adapter模式、Decorator模式、Builder模式、Bridge模式等很相似，它们具有以下相同点：

- 都是通过新类对原有类的封装（以继承或委让的方式），即通过新类访问原有的类。但是它们的目的或行为不同：
 - Proxy模式在客户端访问目标对象前或后，需要进行某些特别的处理。
 - Adapter模式则纯粹为了给目标类做一个封装，使其适合被新的系统使用，即Adapter模式是为了给目标类统一接口。
 - Bridge模式则是为了分离事务内部具有并列属性的抽象与具体行为。
 - Decorator模式则是可以在运行期动态地改变一个对象方法的行为。
- 它们使用的接口方式不同：
- Proxy模式中被代理的目标对象与Proxy类使用同一个接口。
 - Adapter模式是为调用原对象而准备一个新的接口。
 - Bridge模式中的抽象与具体行为分别使用两个不同的接口。
 - Decorator模式除了装饰（Decorator）类与被装饰的类使用同一个接口之外，装饰（Decorator）类的实现还使用委让的方式。

11.3 代理模式总结

有很多问题都可以通过“增加一个中间层”的方式来解决，代理模式就是很好的一个例子，它通过增加了类来实现间接访问业务类的目的。

在使用代理模式的过程中需要注意以下问题：

- 需要考虑代理模式使用的粒度，如：是代理一个类，还是一个功能模块，或者是一个复杂结构，这都是需要认真考虑的问题。
- 代理类的接口与业务对象的接口可以不一样，可按照需求来选择。
- 不要由客户代码指定代理类所要代理的对象，应由代理类自己决定。

代理模式的使用十分广泛，不止在软件方面，在其他领域中，代理模式也屡见不鲜，希望读者好好掌握使用代理模式解决问题的方式，在实际应用中，可以根据需要加以扩展。

第12章 外观模式 (Facade)

Facade模式的意图是：为了给子系统的一组接口提供一个一致的界面，**Facade**模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。¹

现实生活中存在很多复杂的系统，比如要注册一个公司，就必须和银行、市政府、工商局打交道，少则一周多则数月，手续十分复杂。再如汽车系统，也十分复杂，汽车有各种子系统——发动机系统、电力系统、传动系统、空调系统等。像这样的系统使用起来，用户必须了解其各个子系统的详细情况，很显然，这样就会使客户使用系统的复杂度增加，而且如果系统有改变的话，客户的使用步骤也必须更改。在软件设计中，由于模块间的耦合度过高就会出现这样的问题，要解决上述问题，必须要在客户使用的模块和各个子系统之间解耦合。外观模式就是专门用于解决这种问题的。

12.1 顾客的意见——购买大件商品时手续繁多

在超市内购买大件商品时，需要办的手续比较多，比如客户要购买一个背投彩电，需要先通过导购员进行商品的选购，选定了型号之后带着由销售人员开出的购买小票到收银台付款，付款成功后需要到送货处进行住址和联系方式的登记，在合适的时间由送货人员和安装人员上门进行产品的送货安装，经过调试正常后才算购物完成。

这种系统的示意图如图12-1所示。

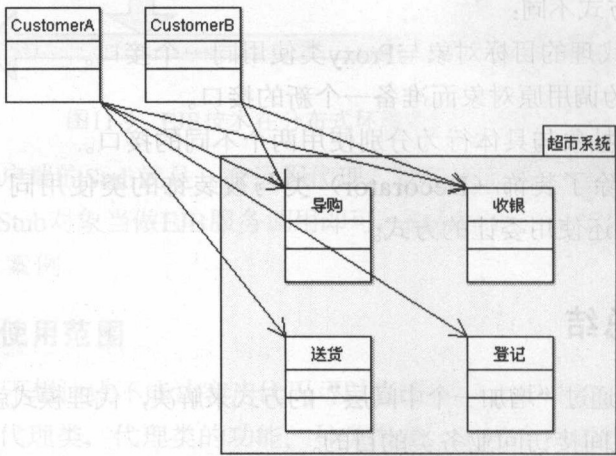


图12-1 超市购物示意图

图12-1展示的这个系统非常复杂，由于不同的客户需要按照自己的需求调用超市的各种系统，这样设计的话会有以下几点不足之处：

- 顾客需要了解每一个子系统。

¹Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. GOF[95]

- 如果一个子系统有更改，则会牵连所有顾客的使用方法。
- 子系统接口复杂，使用烦琐。
- 违反了迪米特法则 (LoD)。

上述的四个缺点中，前三个是使用上的缺陷，第四个涉及设计面向对象软件的法则之一：迪米特法则 (LoD)¹，此法则也称做最少知识原则 (LKP)，其核心思想是：

- 只与你直接的朋友们通信。
- 不要跟“陌生人”说话。
- 每一个软件单位对其他的单位都只了解最少的知识，而且局限于那些与本单位密切相关的软件单位。

在这个法则中比较关键的就是如何确定“朋友”单位，如下原则可供大家选择“朋友”单位：

- 当前对象本身 (this)。
- 以参量形式传入到当前对象方法中的对象。
- 当前对象的实例变量直接引用的对象。
- 当前对象的实例变量如果是一个聚集，那么聚集中的元素也都是朋友。
- 当前对象所创建的对象。

任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”。而对于陌生人就不要与之发生关系。对于一个软件单位与其他单位是否应该发生交互，应该参考以下法则：

- 一个软件实体应当尽可能少地与其他实体发生相互作用。
- 每一个软件单位对其他的单位都只具有最少的知识，而且局限于那些与本单位密切相关的软件单位。

学习了以上各种法则，再来看一下图12-1的设计有什么问题。很显然，客户端其实并不需要得知每一个部门的所有操作方法，比如给导购部门发奖金、送货部门车辆的加油、收银员每日的对账等方法。

这里，使用外观模式设计的结构就可以解决上述的问题。

12.2 外观模式的结构

外观模式解决了客户所要使用的系统中由于子系统过多导致的客户使用困难的情况。

12.2.1 外观模式

外观模式解决由于子系统过多造成的应用复杂性，该方法生成一个和子系统相关的外观类，客户代码使用时调用外观类而不是直接调用子系统，这样就解决了客户代码对各种子系统耦合度高的问题，同时通常会对此外观类的接口设计得比较简单，这样客户代码使用此系统就会很方便。

为了方便学习此模式，下面采用一个汽车系统的例子说明此模式，其类图如图12-2所示。

¹http://en.wikipedia.org/wiki/Law_of_Demeter。

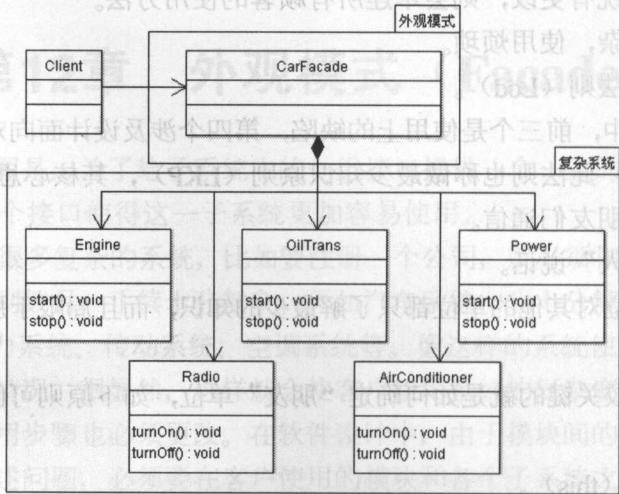


图12-2 外观模式UML类图

图12-2展示的是以汽车系统为例的外观模式的结构，其中用到的主要构成角色如下：

- 外观角色（CarFacade）：此角色提供了接口（类）供客户代码调用来使用系统的功能，通常在使用时此类把客户需要的功能委托给特定的子系统，客户代码不需要直接调用系统方法，只需要使用此类即可，此类在客户和系统之间起到了解耦合的关键作用。
- 子系统角色（Engine、OilTrans、Power...）：实现系统的具体功能，在外观模式中，此类主要提供功能给外观角色使用，但是此类不能依赖外观角色，它是一个下层类，根据分层的原则，不可以依赖上层类（编程中尽量避免出现循环依赖）。

下面先来看一下生成的汽车各个子系统的代码。

引擎子系统：

代码片段1 Engine.java

```
1 package cn.steven.pattern.demo.facade;
2
3 /**
4  * 子系统：引擎
5  */
6 public class Engine {
7
8     public void start() {
9         System.out.println("发动引擎");
10    }
11
12     public void stop() {
13         System.out.println("停止引擎");
14    }
15 }
```

输油系统：

代码片段2 OilTrans.java

```
1 package cn.steven.pattern.demo.facade;
```

```
2
3 /**
4  * 子系统: 输油系统
5  */
6 public class OilTrans {
7
8     public void start() {
9         System.out.println("开启输油系统");
10    }
11
12    public void stop() {
13        System.out.println("关闭输油系统");
14    }
15 }
```

电力系统:

代码片段3 Power.java

```
1 package cn.steven.pattern.demo.facade;
2
3 /**
4  * 子系统: 电力系统
5  */
6 public class Power {
7
8     public void start() {
9         System.out.println("开启电力系统");
10    }
11
12    public void stop() {
13        System.out.println("关闭电力系统");
14    }
15 }
```

代码片段1、代码片段2和代码片段3展示的子系统是需要汽车在启动时就立即启动的子系统,但是还有些系统不是自动启动的,而是在客户需要时再启动,下面就来展示这样的子系统代码。

收音机系统:

代码片段4 Radio.java

```
1 package cn.steven.pattern.demo.facade;
2
3 /**
4  * 子系统: 收音机
5  */
6 public class Radio {
7
8     public void turnOn() {
9         System.out.println("打开收音机");
10    }
11 }
```

```
11
12     public void turnOff() {
13         System.out.println("关闭收音机");
14     }
15 }
```

空调系统:

代码片段5 AirConditioner.java

```
1 package cn.steven.pattern.demo.facade;
2
3 /**
4  * 子系统: 空调系统
5  */
6 public class AirConditioner {
7
8     public void turnOn() {
9         System.out.println("打开空调");
10    }
11
12    public void turnOff() {
13        System.out.println("关闭空调");
14    }
15 }
```

以上各个子系统共同构成了汽车系统，由此可见，如果客户想要使用汽车系统直接调用各个子系统的话，复杂度是很高的，客户必须了解各个子系统的各种功能。如果客户想要使用不同的车型，还需要重新了解新的系统，这使得客户端代码没有稳定性。

为了解决上述问题，我们使用以下外观模式的代码来封装变化:

代码片段6 CarFacade.java

```
1 package cn.steven.pattern.demo.facade;
2
3 /**
4  * 汽车外观类
5  */
6 public class CarFacade {
7
8     /**
9      * 组合各种子系统
10     */
11     private Engine engine = new Engine();
12     private OilTrans oilTrans = new OilTrans();
13     private Power power = new Power();
14     private Radio radio = new Radio();
15     private AirConditioner airConditioner = new AirConditioner();
16     /**
17      * 汽车是否为启动状态, false 未启动, true 已启动
18      */
19     private boolean isStart = false;
```



```

20
21 /**
22  * 使用钥匙发动汽车
23  */
24 public void turnOn() {
25     // 判断是否未启动
26     if (!isStart) {
27         // 启动电力系统
28         power.start();
29         // 启动输油系统
30         oilTrans.start();
31         // 启动引擎
32         engine.start();
33         // 设置状态
34         isStart = true;
35     }
36 }
37
38 /**
39  * 关闭汽车系统
40  */
41 public void turnOff() {
42     // 判断是否已启动
43     if (isStart) {
44         // 停止引擎
45         engine.stop();
46         // 停止输油系统
47         oilTrans.stop();
48         // 停止电力系统
49         power.stop();
50         // 设置状态
51         isStart = false;
52     }
53 }
54
55 /**
56  * 开启收音机
57  */
58 public void turnOnRadio() {
59     // 只有汽车已经启动时才能操作收音机
60     if (isStart) {
61         radio.turnOn();
62     }
63 }
64
65 /**
66  * 关闭收音机
67  */
68 public void turnOffRadio() {
69     // 只有汽车已经启动时才能操作收音机
70     if (isStart) {

```

```
71         radio.turnOff();
72     public void turnOff() {
73     }
74     }
75     /**
76      * 开启空调
77      */
78     public void turnOnAirConditioner() {
79         // 只有汽车已经启动时才能操作空调
80         if (isStart) {
81             airConditioner.turnOn();
82         }
83     }
84
85     /**
86      * 关闭收音机
87      */
88     public void turnOffAirConditioner() {
89         // 只有汽车已经启动时才能操作空调
90         if (isStart) {
91             airConditioner.turnOff();
92         }
93     }
94
95 }
```

代码片段6封装了所有子系统的功能，这样，如果部分子系统有变化，只需要改动此类的代码即可，就不用改动客户端调用此类的代码。

客户端调用代码：

代码片段7 Client.java

```
1 package cn.steven.pattern.demo.facade;
2
3 /**
4  * 外观模式客户端代码
5  */
6 public class Client {
7     public static void main(String[] args) {
8         // 要操作汽车系统需要先实例化外观类
9         CarFacade car = new CarFacade();
10        // 启动
11        car.turnOn();
12        // 使用收音机
13        car.turnOnRadio();
14        car.turnOffRadio();
15        // 使用空调
16        car.turnOnAirConditioner();
17        car.turnOffAirConditioner();
18        // 停止
19        car.turnOff();
```

20 }
21 }

运行结果如图12-3所示。

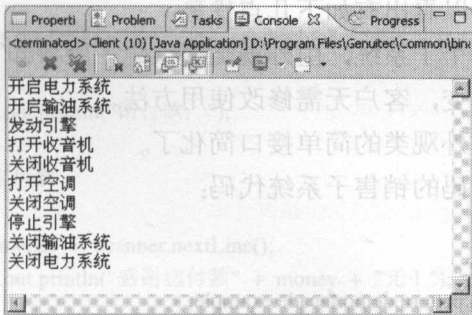


图12-3 Client.java运行结果

由图12-3的运行结果可见，由于客户代码通过调用外观类的功能来使用系统，所以可以通过简单的代码实现复杂的功能，如启动和停止系统，而且可以在系统变化时不必改动客户代码，实现了和各个子系统的低耦合度。

12.2.2 使用外观模式解决顾客的一站式服务

经过了上面外观模式的学习，读者应该可以对图12-1所描述的系统有一定的设计思路了。针对这种有复杂子系统的系统的设计，为了简化客户端代码的使用和隔离子系统的变化，在设计时会对其进行一个外观模式的包装，这样就可以解决上述问题。

客户考虑一个系统时，应该是使用越简单越好。图12-1中有四个子系统，分别是导购、收银、送货和登记系统。在超市中，这些系统的工作地点通常不在同一个地方，客户要使用的话需要打听各个系统的工作位置以及亲自过去办理业务，这一过程要通过复杂的操作才能完成。

在以下的设计中，引入了一个“购物助手”的类，此类可以带领客户完成购物的全过程，顾客需要购物时只需要和购物助手互动即可，从而极大地方便了顾客，其类图如图12-4所示。

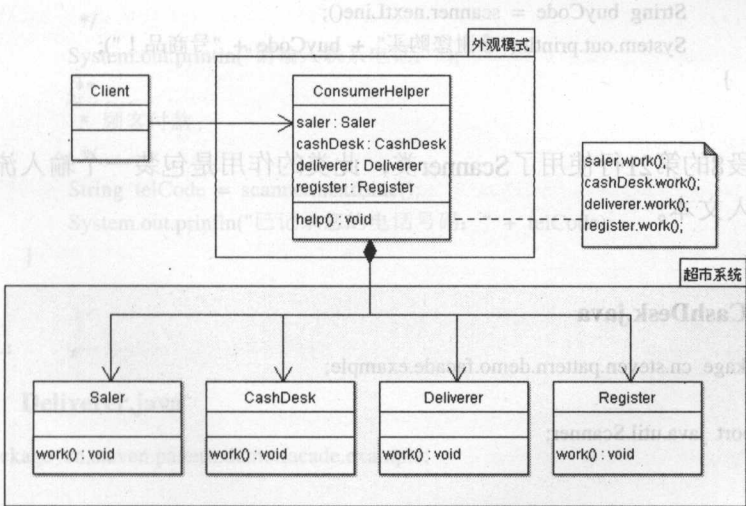


图12-4 超市购物的外观模式UML类图

由图12-4可以看出使用外观模式设计系统的基本思路与图12-1的区别是在原有系统的基础上增加了一个外观类ConsumerHelper，客户端代码转而调用此类来替代原有的直接操作子系统类。

经过了这样的设计，可以看出有如下几点优势：

- 顾客无需了解任何一个子系统，由外观类提供统一服务即可。
- 如果一个子系统有改变，客户无需修改使用方法。
- 子系统的复杂接口被外观类的简单接口简化了。

下面展示的是子系统代码的销售子系统代码：

代码片段8 Saler.java

```
1 package cn.steven.pattern.demo.facade.example;
2
3 import java.util.Scanner;
4
5 /**
6  * 导购
7  */
8 public class Saler {
9     /**
10      * 工作方法
11      */
12     public void work(Scanner scanner) {
13         /**
14          * 售货员给出选择
15          */
16         System.out.println("请输入要买的商品编号：");
17         System.out.println("1.电视    2.冰箱    3.洗衣机    4.空调");
18         /**
19          * 顾客购买
20          */
21         String buyCode = scanner.nextLine();
22         System.out.println("感谢您购买" + buyCode + "号商品！");
23     }
24 }
```

注意代码片段8的第21行使用了Scanner类，此类的作用是包装一个输入流，在这里的作用是获得客户的输入文本。

收款子系统：

代码片段9 CashDesk.java

```
1 package cn.steven.pattern.demo.facade.example;
2
3 import java.util.Scanner;
4
5 /**
6  * 收款
7  */
8 public class CashDesk {
```

```

9      /**
10     * 工作方法
11     */
12     public void work(Scanner scanner) {
13         /**
14         * 收款员提示付款
15         */
16         System.out.println("请付款: ");
17         /**
18         * 顾客付款
19         */
20         String money = scanner.nextLine();
21         System.out.println("感谢您付款" + money + "元!");
22     }
23 }

```

注册子系统:

代码片段10 Register.java

```

1 package cn.steven.pattern.demo.facade.example;
2
3 import java.util.Scanner;
4
5 /**
6  * 登记
7  */
8 public class Register {
9     /**
10    * 工作方法
11    */
12    public void work(Scanner scanner) {
13        /**
14        * 提示登记
15        */
16        System.out.println("请输入联系电话: ");
17        /**
18        * 顾客付款
19        */
20        String telCode = scanner.nextLine();
21        System.out.println("已记录您的电话号码: " + telCode);
22    }
23 }

```

送货子系统:

代码片段11 Deliverer.java

```

1 package cn.steven.pattern.demo.facade.example;
2
3 import java.util.Scanner;
4
5 /**

```

```

6  * 送货
7  */
8  public class Deliverer {
9      /**
10     * 工作方法
11     */
12     public void work(Scanner scanner) {
13         /**
14         * 提示给出送货时间
15         */
16         System.out.println("请给出送货时间: ");
17         /**
18         * 顾客付款
19         */
20         String date = scanner.nextLine();
21         System.out.println("送货时间: " + date);
22     }
23 }

```

下面展示的就是构成外观模式的关键类——购物助手类:

代码片段12 ConsumerHelper.java

```

1  package cn.steven.pattern.demo.facade.example;
2
3  import java.util.Scanner;
4
5  /**
6   * 购物助手
7   */
8  public class ConsumerHelper {
9
10     /**
11     * 销售人员
12     */
13     private Saler saler = new Saler();
14     /**
15     * 收银员
16     */
17     private CashDesk cashDesk = new CashDesk();
18     /**
19     * 注册员
20     */
21     private Register register = new Register();
22     /**
23     * 送货员
24     */
25     private Deliverer deliverer = new Deliverer();
26
27     /**
28     * 帮助方法
29     */

```



```

30 public void help() {
31     System.out.println("购物助手说：开始购物！");
32     /**
33      * 客户端输入的扫描器
34      */
35     Scanner scanner = new Scanner(System.in);
36
37     /**
38      * 引导顾客去购物
39      */
40     saler.work(scanner);
41     /**
42      * 引导顾客去付款
43      */
44     cashDesk.work(scanner);
45     /**
46      * 引导顾客去登记
47      */
48     register.work(scanner);
49     /**
50      * 引导顾客去安排送货
51      */
52     deliverer.work(scanner);
53     System.out.println("购物助手说：购物结束！");
54 }
55 }
    
```

由代码片段12第31行~第53行可见，此助手类调用了多种子系统的功能，但是其接口的使用却很简单。注意代码第35行创建了一个扫描器对象用于获得客户输入。

客户端代码：

代码片段13 Client.java

```

1 package cn.steven.pattern.demo.facade.example;
2
3 /**
4  * 客户端代码
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10        /**
11         * 找到购物助手进行购物
12         */
13        ConsumerHelper consumerHelper = new ConsumerHelper();
14        consumerHelper.help();
15    }
16
17 }
    
```

运行结果如图12-5所示。

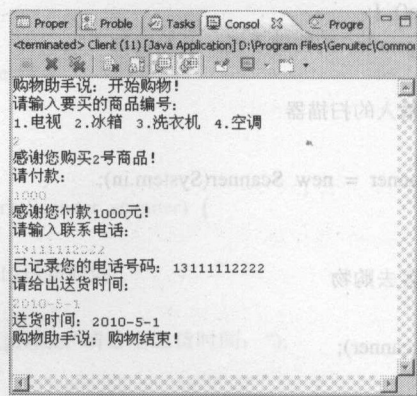


图12-5 购物系统运行结果

由图12-5可见，虽然代码片段13中只有第13行和第14行进行了外观模式的调用，但是却使用了全部的子系统功能，极大方便了客户对于系统的使用。

12.2.3 外观模式在JDK中的实例

JDK集合框架是一个在编程中经常用到的类集合，它由Collection接口牵头，Set、List和Map承担主要接口，用数据结构Tree、Array、Hash、Link作为具体实现手段，组成了一个在功能和性能上相当不错的框架。整个框架由主体结构 and 辅助结构组成，主体结构形成了框架基础，辅助结构为使用框架提供了便利，如表12-1所示。

表12-1 JDK集合框架表

集合 框架		实现类			
		Hash Table	Resizable Array	Balanced Tree	Linked List Hash Table + Linked List
接口	Set	HashSet		TreeSet	LinkedHashSet
	List		ArrayList		LinkedList
	Deque		ArrayDeque		LinkedList
	Map	HashMap		TreeMap	LinkedHashMap

表格12-1展示了集合框架中的大部分接口和实现类，由此可见，集合框架的子系统是十分复杂的，客户端代码在编写时也会十分复杂，比如客户需要对ArrayList、TreeSet和ArrayDeque的对象进行排序操作，就必须分别了解其实现细节并操作，对于不同的集合对象的操作代码不能重用。

为了解决此问题，JDK中设计了一个外观类——Collections来帮助我们方便地操作不同的集合类，其功能说明如下：

java.util.Collections

此类完全由在Collection上进行操作或返回Collection的静态方法组成。它包含在 Collec-

¹详细内容参见<http://java.sun.com/javase/6/docs/technotes/guides/collections/overview.html>。

tion上操作的多态算法, 即“包装器”, 包装器返回由指定Collection支持的新Collection, 以及少数其他内容。

如果为此类的方法所提供的Collection或类对象为null, 则这些方法都将抛出NullPointerException。

此类中所含多态算法的文档通常包括对实现的简短描述, 应该将这类描述视为实现注意事项, 而不是规范的一部分。实现者应该可以随意使用其他算法替代, 只要遵循规范本身即可(例如, sort使用的算法不一定是合并排序算法, 但它必须是稳定的)。

此类中包含的“破坏性”算法, 即可修改其所操作的Collection的算法, 这种算法被指定在Collection不支持适当的可变基元(比如Set方法)时抛出UnsupportedOperationException。如果调用不会对Collection产生任何影响, 那么这些算法也可能(但不要求)抛出此异常。例如, 在已经排序的、不可修改的列表上调用Sort方法可能会(也可能不会)抛出UnsupportedOperationException。

Collections类包含了对框架中集合的普遍性操作, 这些操作都被实现为静态函数, 它们分为以下几类:

- 最大、最小函数, 对于某个集合, 寻找最大、最小成员(或键成员), 要求成员都必须实现comparable接口。
- 生成不可更改的singleton集合(singleton集合为包含一个成员的集合)。
- 对list进行排序、重组、倒序操作。
- 对list进行填充、批量置换和快速搜索操作。
- 对集合进行同步化, 以便适应多线程环境。
- 对集合进行不可更改修饰, 返回不可更改的集合视图。

注意Collections类中所有的方法都是静态的, 这就方便了程序员把它当成一个助手类(和超市的购物助手相似)来使用, 如下就是使用此类的代码示例:

代码片段14 CollectionTest.java

```

1 package cn.steven.pattern.demo.facade;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5 import java.util.Collections;
6 import java.util.List;
7
8 /**
9  * Collections外观类的使用
10  */
11 public class CollectionTest {
12     /**
13      * 客户代码
14      */
15     public static void main(String[] args) {
16         /**
17          * 创建一个集合对象
18          */

```



```

19      Collection<String> c = new ArrayList<String>();
20      c.add("mary");
21      c.add("steven");
22      c.add("john");
23      CollectionTest.print(c);
24      /**
25       * 使用二分法查找指定数据的位置
26       */
27      System.out.println("steven 出现的位置: "
28          + Collections.binarySearch((List<String>) c, "steven"));
29      /**
30       * 字典顺序排序List
31       */
32      Collections.sort((List<String>) c);
33      CollectionTest.print(c);
34      /**
35       * 洗牌算法
36       */
37      Collections.shuffle((List<String>) c);
38      CollectionTest.print(c);
39      /**
40       * 取最大值
41       */
42      System.out.println("max:" + Collections.max(c));
43      /**
44       * 创建一个线程安全的集合对象
45       */
46      Collection<String> synC = Collections.synchronizedCollection(c);
47      /**
48       * 返回一个不可变的制定数据列表
49       */
50      CollectionTest.print(Collections.singletonList("newGirl"));
51      /**
52       * 替换值
53       */
54      Collections.replaceAll((List<String>) c, "mary", "kitty");
55      CollectionTest.print(c);
56      /**
57       * 替换所有值
58       */
59      Collections.fill((List<String>) c, "girl");
60      CollectionTest.print(c);
61      }
62
63      /**
64       * 打印集合对象
65       */
66      public static void print(Collection<?> c) {
67          for (Object object : c) {
68              System.out.print(object + " ");
69          }

```

```
70      System.out.println();
71  }
72 }
```

代码片段14的运行结果如图12-6所示。

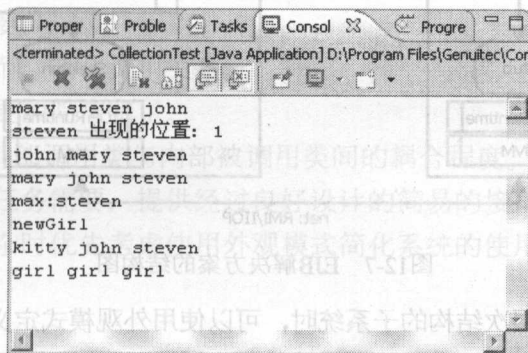


图12-6 CollectionTest运行结果

可见，使用了Collectoins作为集合框架的外观类后，使用各种集合类型就变得容易了，程序员在使用这个框架时也就不需要逐个学习各个具体类中的方法了，这就极大地方便了开发。

JavaEE中也推荐使用外观模式来封装对EJB的调用。为了执行一个典型用例的商业逻辑，多个服务器端对象（如session或entity bean）通常需要被存取和进行可能的修改。问题是session和entity bean的多个细粒度调用增加了多次网络调用（而且可能是多个事务的）的网络开销，并且导致产生了维护性差的代码，因为数据存取和工作流、商业逻辑在客户端之间分散了。

一个典型的转账过程，可能需要操作两个以上的entity bean，这几步操作需要在同一个事务中进行，客户如果直接调用entity bean，不仅增加了网络的开销，而且事务的控制代码由客户代码自己完成的话容易产生安全问题。总之，这样设计的缺点如下：

- 高的网络开销。
- 差的并发性。
- 高耦合。
- 差的可复用性。
- 差的可维护性。
- 差的开发角色的分离。

解决方案是用一个叫做session facade的session bean层来封装entity bean层。客户端应该只能存取session bean，不能存取entity bean，结构图如图12-7所示。

session facade模式把传统的facade模式的好处应用到EJB，完全对客户端隐藏服务器上的对象模型，用session bean层作为客户端的单独存取点。

12.2.4 外观模式的使用范围及优点

外观模式的使用范围：

- 当你要为一个复杂子系统提供一个简单接口时。
- 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入外观模式将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

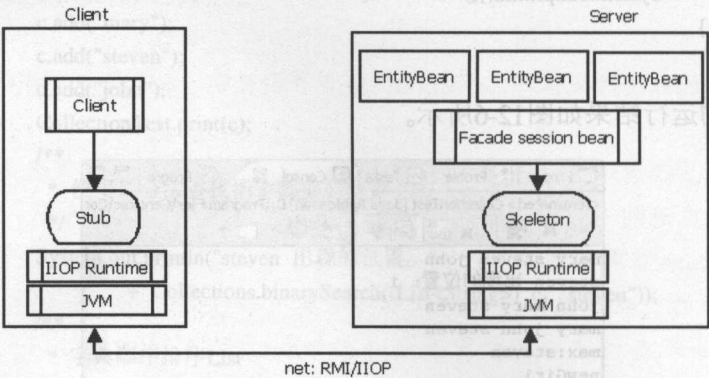


图12-7 EJB解决方案的结构图

• 当你需要构建一个层次结构的子系统时，可以使用外观模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过外观模式进行通信，从而简化了它们之间的依赖关系。

外观模式的优点：

- 对客户屏蔽子系统组件，简化了子系统的使用接口，因而减少了客户处理的对象的数目并使得整个系统使用起来更加方便。
- 实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。松耦合关系使得子系统的组件变化不会影响到它的客户。外观模式有助于建立层次结构系统，也有助于对对象之间的依赖关系分层。外观模式可以消除复杂的循环依赖关系。这一点在客户程序与子系统是分别实现的时候尤为重要。在大型件系统中降低修改时的编译依赖性至关重要，在子系统类改变时，应尽量减少重编译的工作以节省时间。用外观模式可以降低编译依赖性，限制重要系统中较小的变化所需的重编译工作。外观模式同样也有利于简化系统在不同平台之间的移植过程，因为编译个别子系统一般不需要编译所有其他的子系统。

• 外观模式并不限制客户代码直接使用子系统类，因此你可以让客户程序在系统易用性和通用性之间加以选择。

12.2.5 与其他模式的关系

结构型模式的类组成方式都比较相似，但是侧重点却不一样：

- Facade模式注重简化接口。
- Adapter模式注重转换接口。
- Bridge模式注重分离接口（抽象）与其实现。
- Decorator模式注重在稳定接口的前提下为对象扩展功能。

在选用结构性模式时，需要首先明确使用意图，再做选择。

12.3 外观模式总结

外观模式为一组具有类似功能的类群，比如类库，子系统等，提供一个一致的简单界面。这个一致的简单界面被称做Facade。Facade类的设计通常会使用很简单的接口，它会调用各个

子系统来实现用户功能，通常用户不需要直接调用子系统而调用外观类即可。

在应用系统中，为了实现某些具有复杂功能的模块或子系统时，往往需要为其设计和实现很多很小的类，也就是说，该模块或子系统是由一组具有类似功能的类群组合而成。这样一来，怎么调用这些类就成了问题。**Facade**就是这样一种模式，它设计一个被称为**facade**的类，该类提供一个简单的调用接口：

- 隐藏具体的实现细节，简化调用关系。
- 使得调用方的代码更加简洁明了。
- 通过**Facade**，降低外部调用类与内部被调用类间的耦合程度。
- 可以为每个不同的任务需要，提供经过良好设计的简易的接口。

应该在子系统比较复杂时优先考虑使用外观模式简化系统的使用。

13.2 装饰模式的结构

装饰模式的实现方法，在下面的讲解中，我们采用一个打印发票的例子来说明。在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

现实的发票有三部分组成：头部分、主体部分、尾部分。不同的对象打印出来的头和尾是不一样的，但是主体部分是一样的，要实现这样的需求，就应该用装饰模式。

装饰模式的实现方法，在下面的讲解中，我们采用一个打印发票的例子来说明。在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

装饰模式的实现方法，在下面的讲解中，我们采用一个打印发票的例子来说明。在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

装饰模式的实现方法，在下面的讲解中，我们采用一个打印发票的例子来说明。在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

装饰模式的实现方法，在下面的讲解中，我们采用一个打印发票的例子来说明。在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

装饰模式的实现方法，在下面的讲解中，我们采用一个打印发票的例子来说明。在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

装饰模式的实现方法，在下面的讲解中，我们采用一个打印发票的例子来说明。在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

第13章 装饰模式 (Decorator)

装饰模式可以动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式比生成子类更为灵活¹。

在日常生活中，经常会遇到一些情况，比如不论一幅画有没有画框都可以挂在墙上，但是通常仍会使用画框，并且实际上是将画框挂在墙上。在将画框挂在墙上之前，画上可以罩上玻璃，装到框子里；这时画、玻璃和画框就形成了一个物体。装饰上了这些东西之后，我们还会称它是一幅画。

在软件系统中，有时候我们会使用继承来扩展对象的功能，但是由于继承为类型引入了静态特质，这种扩展方式缺乏灵活性，并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能的组合）会导致更多子类的膨胀。如何使“对象功能的扩展”能够根据需要来动态地实现，同时避免“扩展功能的增多”带来的子类膨胀问题，并使得任何“功能扩展变化”所导致的影响降为最低？这就是装饰模式要解决的问题。

有一款叫做《模拟更衣室》的小游戏，可以给游戏人物动态添加衣服、裤子、首饰等物品。由于有很多不同的衣物，那么对象也有不同的功能扩展，要达到这种效果必须使用装饰模式。如果利用继承的方式来扩展对象的功能，将会导致类的爆炸，而且继承的扩展是静态的，不能根据需要动态扩展，所以装饰模式是以客户透明的方式动态地对对象功能进行的扩展，也就是对这个对象附加新的职责。

13.1 如何解决销售人员的能力固化问题

在超市中，所有的销售人员都有自己的工作职责，在入职时，他们经培训学习了所需的技能，如销售技巧、礼仪规范、法规约定等。经过长时间的工作，销售人员的能力慢慢固化了下来，随着超市的发展，公司决定给某些特定的销售人员再进行一次培训提高他们的工作能力，以提高服务质量，提升公司的竞争力。

对于化妆品销售人员，公司决定在其销售过程之前加一个帮助客户试用的过程，并对其进行了培训。在所有的销售人员培训中，进行了销售前摆货架、销售后打扫卫生以及帮助有困难顾客进行购物等的技能培训。

面向对象的知识提供给我们一种给类增加能力的办法，就是继承。由以上需求我们可以设计出以下销售人员的UML类图，如图13-1所示。

经过了这些改进后，会发现原来清晰的类关系变得非常复杂。要给类增加功能，可以使用继承来扩展对象的功能，但是由于继承为类型引入了静态特质，使得这种扩展方式缺乏灵活性；并且随着子类的增多（扩展功能的增多，在此例中就是销售人员能力的增多），各种子类的组合（扩展功能的组合，在此例中就是各种技能的组合）会导致更多子类的膨胀，最终，整个程

¹Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.GOF[95]

序会因为不可控制的类爆炸而使得设计失败。

设计模式中的装饰模式就是用来解决此类问题的。

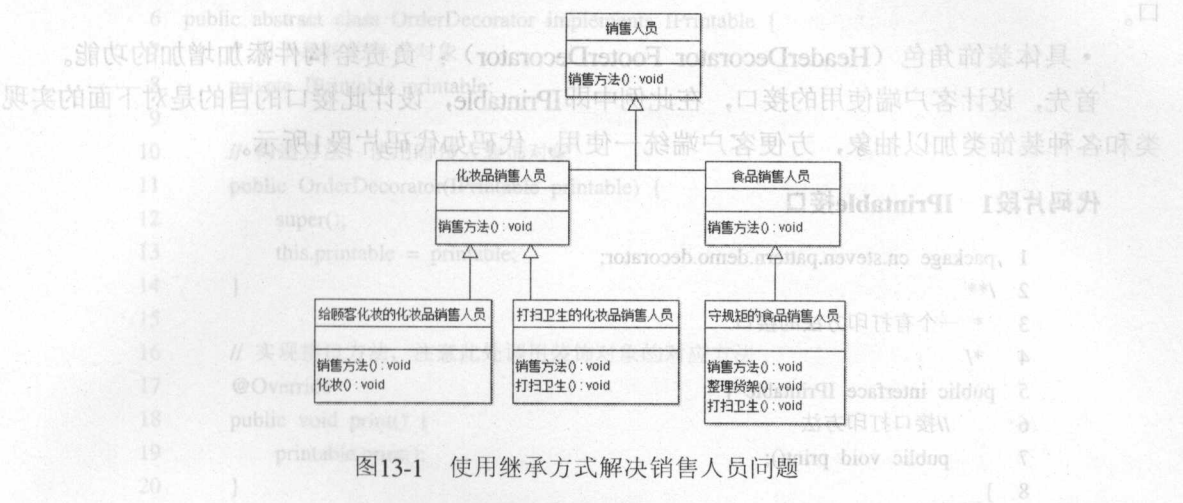


图13-1 使用继承方式解决销售人员问题

13.2 装饰模式的结构

装饰模式可以再无需创建子类的情况下扩展类的功能。

13.2.1 装饰模式

为了说明装饰模式的实现方法，在下面的讲解中，我们采用一个打发票的例子来进行讲解，在这一过程中我们通过不断地增加打印内容来说明装饰模式的用法。

现实的发票有三部分组成：头部分，主体部分（数据部分），尾部分；不同的对象打印出来的头和尾是不一样的，但是主体部分是一样，要实现这样的需求，就应该采用装饰模式，下面是UML图，如图13-2所示。

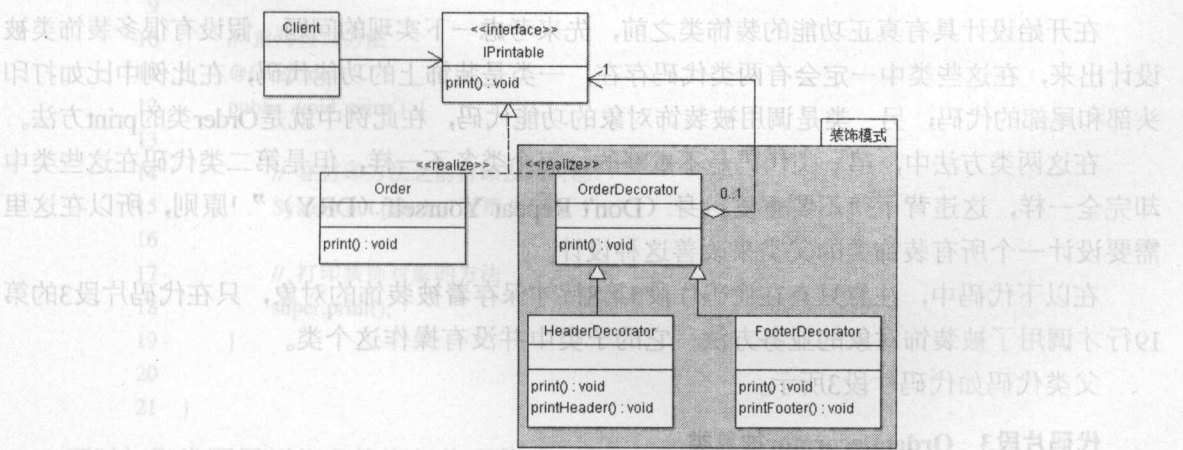


图13-2 打印发票的UML类图

参照图13-2中的类图可知装饰模式的各种角色有：

- 抽象构件角色（IPrintable）：定义一个抽象接口，以规范准备接收附加责任的对象。

- 具体构件角色（Order）：这是被装饰者，定义一个将要被装饰模式增加功能的类。
- 装饰角色（OrderDecorator）：持有一个构件对象的实例，并定义了抽象构件定义的接口。
- 具体装饰角色（HeaderDecorator FooterDecorator）：负责给构件添加增加的功能。

首先，设计客户端使用的接口，在此例中即 **IPrintable**，设计此接口的目的是对下面的实现类和各种装饰类加以抽象，方便客户端统一使用，代码如代码片段1所示。

代码片段1 IPrintable接口

```
1 package cn.steven.pattern.demo.decorator;  
2 /**  
3  * 一个有打印方法的接口  
4  */  
5 public interface IPrintable {  
6     //接口打印方法  
7     public void print();  
8 }
```

设计一个实现类可供客户端使用，代码如代码片段2所示。

代码片段2 Order发票打印类

```
1 package cn.steven.pattern.demo.decorator;  
2 /**  
3  * 发票  
4  */  
5 public class Order implements IPrintable {  
6     //实现接口中的方法  
7     public void print(){  
8         System.out.println("发票的主体部分");  
9     }  
10 }
```

在开始设计具有真正功能的装饰类之前，先来考虑一下实现的问题。假设有很多装饰类被设计出来，在这些类中一定会有两类代码存在，一类是装饰上的功能代码，在此例中比如打印头部和尾部的代码；另一类是调用被装饰对象的功能代码，在此例中就是 **Order** 类的 **print** 方法。

在这两类方法中，第一类代码是不重复的，每个类各不一样，但是第二类代码在这些类中却完全一样，这违背了“不要重复自身（Don't Repeat Yourself (DRY)）”¹原则，所以在这里需要设计一个所有装饰类的父类来改善这种设计。

在以下代码中，注意只有在代码片段3第8行才保存着被装饰的对象，只在代码片段3的第19行才调用了被装饰对象的业务方法。它的子类中并没有操作这个类。

父类代码如代码片段3所示。

代码片段3 OrderDecorator抽象类

```
1 package cn.steven.pattern.demo.decorator;  
2
```

¹http://en.wikipedia.org/wiki/Don%27t_repeat_yourself.

```
3 /**
4  * 发票装饰抽象类
5  */
6 public abstract class OrderDecorator implements IPrintable {
7     // 用于装饰的私有对象
8     private IPrintable printable;
9
10    // 构造方法, 使用时传入装饰对象
11    public OrderDecorator(IPrintable printable) {
12        super();
13        this.printable = printable;
14    }
15
16    // 实现接口方法, 注意此处调用装饰对象的对应方法
17    @Override
18    public void print() {
19        printable.print();
20    }
21
22 }
```

可以打印发票头部信息的类如代码片段4所示。

代码片段4 HeaderDecorator装饰类

```
1 package cn.steven.pattern.demo.decorator;
2
3 public class HeaderDecorator extends OrderDecorator {
4
5     // 构造方法, 传入装饰对象
6     public HeaderDecorator(IPrintable printable) {
7         super(printable);
8     }
9
10    // 重写接口方法
11    @Override
12    public void print() {
13
14        // 在打印方法之前打印发票头部
15        System.out.println("发票头部");
16
17        // 打印装饰对象的方法
18        super.print();
19    }
20
21 }
```

可以打印发票尾部信息的类如代码片段5所示。

代码片段5 FooterDecorator装饰类

```
1 package cn.steven.pattern.demo.decorator;
```

```
2
```

```

3 public class FooterDecorator extends OrderDecorator {
4     // 构造方法，传入装饰对象
5     public FooterDecorator(IPrintable printable) {
6         super(printable);
7     }
8
9     // 实现接口方法，注意此处调用装饰对象的对应方法
10    @Override
11    public void print() {
12
13        // 打印装饰对象的方法
14        super.print();
15
16        // 在打印方法之后打印发票尾部
17        System.out.println("发票尾部");
18    }
19
20 }

```

客户端代码如代码片段6所示。

代码片段6 Client类

```

1 package cn.steven.pattern.demo.decorator;
2
3 /**
4  * 装饰模式客户端
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10        // 测试原始类
11        IPrintable order = new Order();
12        order.print();
13        System.out.println("-----打印完毕-----");
14
15        // 测试装饰头部的类
16        IPrintable headerOrder = new HeaderDecorator(new Order());
17        headerOrder.print();
18        System.out.println("-----打印完毕-----");
19
20        // 测试装饰头部和尾部的类
21        IPrintable headerAndFooterOrder = new HeaderDecorator(
22            new FooterDecorator(new Order()));
23        headerAndFooterOrder.print();
24        System.out.println("-----打印完毕-----");
25    }
26
27 }

```


请注意代码片段6的第16行和第21行中的装饰模式的典型用法，它看起来像是把一个对象一层一层包装起来一样，这也正是装饰模式的别名“包装器 (Wrapper)”的由来。代码的运行结果如图13-3所示。

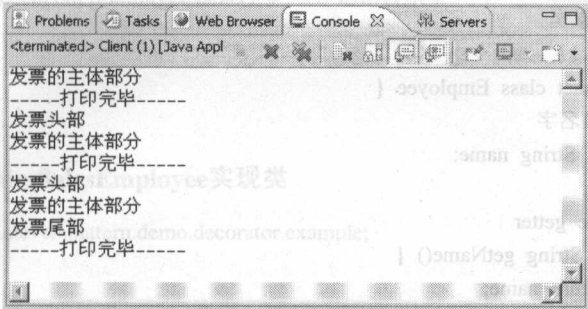


图13-3 打印发票的装饰模式的运行结果

可见，经过了装饰类的包装后，被装饰的对象具备了原来没有的功能，而且，任意组合各种装饰方法是十分方便的，由于装饰模式没有改变被装饰类的接口，所以客户端的使用代码并没有改变。

13.2.2 使用装饰模式解决销售人员问题

经过了上一节的分析，现在就可以使用装饰模式来解决销售人员能力固化的问题了。经过研究，可以得到以下UML类图，如图13-4所示。

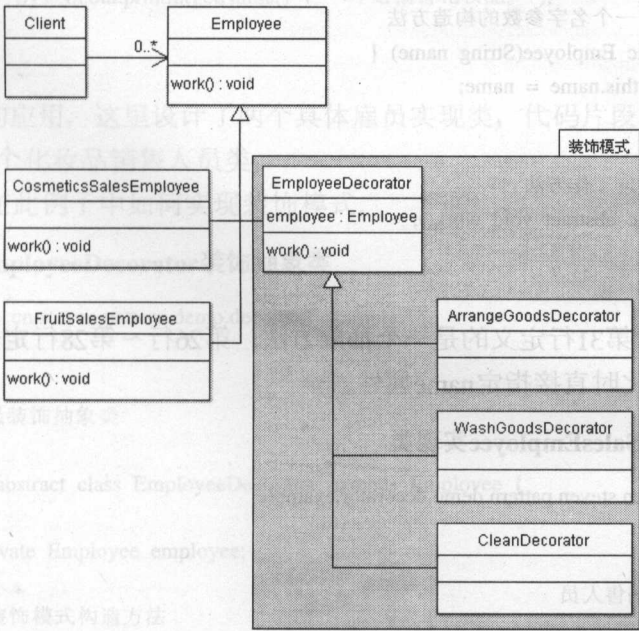


图13-4 解决销售人员能力固化的UML类图

图13-4与图13-2的不同之处在于此处使用抽象类作为整个架构的根，而上例中使用的是接口。下面先来看一下本例中的基础类。

代码片段7 Employee抽象类

```

1 package cn.steven.pattern.demo.decorator.example;
2
3 /**
4  * 雇员抽象类
5  */
6 public abstract class Employee {
7     // 雇员名字
8     private String name;
9
10    // name getter
11    public String getName() {
12        return name;
13    }
14
15    // name setter
16    public void setName(String name) {
17        this.name = name;
18    }
19
20    // 无参构造方法
21    public Employee() {
22        super();
23    }
24
25    // 带一个名字参数的构造方法
26    public Employee(String name) {
27        this.name = name;
28    }
29
30    // 抽象工作方法
31    public abstract void work();
32
33 }

```

注意代码片段7中第31行定义的是一个抽象方法，第26行~第28行定义了一个带参数的构造方法，用于在实例化时直接指定name属性。

代码片段8 FruitSalesEmployee实现类

```

1 package cn.steven.pattern.demo.decorator.example;
2
3 /**
4  * 水果销售人员
5  */
6 public class FruitSalesEmployee extends Employee {
7
8     // 构造方法
9     public FruitSalesEmployee(String name) {
10         super(name);
11     }

```

```

12 // 重写父类的抽象方法
13 @Override
14 public void work() {
15     System.out.println(getName() + " 开始销售水果。");
16 }
17
18
19 }

```

代码片段9 CosmeticsSalesEmployee实现类

```

1 package cn.steven.pattern.demo.decorator.example;
2
3 /**
4  * 化妆品销售人员
5  */
6 public class CosmeticsSalesEmployee extends Employee {
7     // 构造方法
8     public CosmeticsSalesEmployee(String name) {
9         super(name);
10    }
11
12    // 重写父类的抽象方法
13    @Override
14    public void work() {
15        System.out.println(getName() + " 开始销售化妆品。");
16    }
17 }

```

为了演示完整的应用，这里设计了两个具体雇员实现类，代码片段8是一个水果销售人员类，代码片段9是一个化妆品销售人员类。

下面来看一下在此例子中如何实现装饰模式。

代码片段10 EmployeeDecorator装饰抽象类

```

1 package cn.steven.pattern.demo.decorator.example;
2
3 /**
4  * 雇员装饰抽象类
5  */
6 public abstract class EmployeeDecorator extends Employee {
7
8     private Employee employee;
9
10    //装饰模式构造方法
11    public EmployeeDecorator(Employee employee) {
12        this.employee = employee;
13    }
14
15    //此处需要重写，委托给被装饰的对象
16    @Override
17    public String getName() {

```



```

18 return employee.getName();
19 }
20
21 // 重写父类的抽象方法
22 @Override
23 public void work() {
24     employee.work();
25 }
26
27 }

```

在代码片段10中，注意第11行~第13行为典型的装饰模式构造方法，使用此方法传入被装饰对象后，将其保存在属性中（第8行），由于装饰子类需要用到被装饰对象的getName方法，所以第17行~第19行对此方法进行了委托。

下面来看一下装饰模式的三个具体实现类。

代码片段11 CleanDecorator装饰实现类

```

1 package cn.steven.pattern.demo.decorator.example;
2
3 /**
4  * 工作完成后需要打扫
5  */
6 public class CleanDecorator extends EmployeeDecorator {
7
8     // 构造方法
9     public CleanDecorator(Employee employee) {
10         super(employee);
11     }
12
13     // 重写父类的方法
14     @Override
15     public void work() {
16         super.work();
17         // 前提是工作完后打扫，所以要放在这个位置
18         clean();
19     }
20
21     // 增加一个工作方法
22     public void clean() {
23         System.out.println(this.getName() + " 开始打扫!");
24     }
25
26 }

```

注意CleanDecorator装饰实现类中把增加功能的执行顺序放在了工作方法之后。

代码片段12 ArrangeGoodsDecorator装饰实现类

```

1 package cn.steven.pattern.demo.decorator.example;
2
3 /**

```

```
4 * 工作前整理货物
```

```
5 */
```

```
6 public class ArrangeGoodsDecorator extends EmployeeDecorator {
```

```
7
```

```
8 // 构造方法
```

```
9 public ArrangeGoodsDecorator(Employee employee) {
```

```
10     super(employee);
```

```
11 }
```

```
12
```

```
13 // 重写父类的方法
```

```
14 @Override
```

```
15 public void work() {
```

```
16     // 前提是工作前整理货物，所以要放在这个位置
```

```
17     arrange();
```

```
18     super.work();
```

```
19 }
```

```
20
```

```
21 // 增加一个工作方法
```

```
22 public void arrange() {
```

```
23     System.out.println(this.getName() + " 开始整理货架!");
```

```
24 }
```

```
25
```

```
26 }
```

注意ArrangeGoodsDecorator装饰实现类中把增加功能的执行顺序放在了工作方法之前。

代码片段13 WashGoodsDecorator装饰实现类

```
1 package cn.steven.pattern.demo.decorator.example;
```

```
2
```

```
3 /**
```

```
4 * 工作前清洗商品
```

```
5 */
```

```
6 public class WashGoodsDecorator extends EmployeeDecorator {
```

```
7
```

```
8 // 构造方法
```

```
9 public WashGoodsDecorator(Employee employee) {
```

```
10     super(employee);
```

```
11 }
```

```
12
```

```
13 // 重写父类的方法
```

```
14 @Override
```

```
15 public void work() {
```

```
16     // 前提是工作前清洗商品，所以要放在这个位置
```

```
17     wash();
```

```
18     super.work();
```

```
19 }
```

```
20
```

```
21 // 增加一个工作方法
```

```
22 public void wash() {
```

```
23     System.out.println(this.getName() + " 开始清洗商品!");
```

```
24 }
```

25

26 }

注意WashGoodsDecorator装饰实现类中把增加功能的执行顺序放在了工作方法之前。

至此，已经完成了整个装饰模式的编码工作，现在所有可变的附加功能已经分散到了各个独立的类中，如果需要使用，只需要把这些类进行组合使用即可。

代码片段14 Client客户使用类

```
1 package cn.steven.pattern.demo.decorator.example;
2
3 public class Client {
4
5     public static void main(String[] args) {
6         // 定义一个销售水果人员
7         Employee fruitSaler = new FruitSalesEmployee("宋江");
8         // 定义一个化妆品销售人员
9         Employee cosmeticsSaler = new CosmeticsSalesEmployee("孙二娘");
10
11         System.out.println("log:本职工作开始");
12         fruitSaler.work();
13         cosmeticsSaler.work();
14
15         System.out.println("log:装饰工作开始");
16
17         // 使用装饰器
18         // 根据不同的需要进行装饰器的组合
19         fruitSaler = new CleanDecorator(new WashGoodsDecorator(
20             new WashGoodsDecorator(fruitSaler)));
21         fruitSaler.work();
22
23         cosmeticsSaler = new ArrangeGoodsDecorator(new CleanDecorator(
24             cosmeticsSaler));
25         cosmeticsSaler.work();
26
27     }
28 }
```

注意代码片段14中的第19行和第23行中是怎样对装饰类进行包装使用的。使用者可以根据自己的需要进行任意组合。

运行结果如图13-5所示。

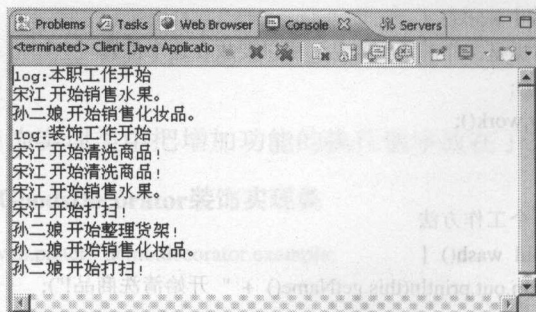


图13-5 运行结果

13.2.3 装饰模式在JDK中的实例

在JDK中，有很多使用装饰模式的案例，如java.io包中对流的处理。

Java中流 (Stream) 是一种有序的字节数据对象。流又分为输入流 (InputStream) 和输出流 (OutputStream)。输入流从外部资源 (文件，内存，socket等) 读入字节数据到Java对象；输出流则把Java对象 (字节数据等) 写入到外部资源。

所有Java I/O都可归类为以下两种：

- 字节数据输入输出I/O。
- 文字列数据输入输出I/O。

在下例中，只研究InputStream类的装饰模式，理解了这一种设计方式，其他的就很好理解了，因为它们的设计方式是很相似的。

下面展示的是InputStream类的直接继承子类：

- ByteArrayInputStream
- FileInputStream
- FilterInputStream
- ObjectInputStream
- PipedInputStream
- SequenceInputStream
- StringBufferInputStream

根据输入流的源的类型，可以将这些流类分成两种，即原始流处理器 (Original Stream) 和链接流处理器 (Wrapper Stream)。

原始流处理器接收一个Byte数组对象，String对象，FileDescriptor对象或者不同类型的流源对象。原始流处理器包括以下四种：

- **ByteArrayInputStream**：为多线程的通信提供缓冲区操作功能，接收一个Byte数组作为流的源。
- **FileInputStream**：建立一个与文件有关的输入流。接收一个File对象作为流的源。
- **PipedInputStream**：可以与PipedOutputStream配合使用，用于读入一个数据管道的数据，接收一个PipedOutputStream作为源。
- **StringBufferInputStream**：将一个字符串缓冲区转换为一个输入流，接收一个String对象作为流的源 (JDK帮助文档上说明：已过时。此类未能正确地将字符转换为字节。从JDK 1.1版本开始，从字符串创建流的首选方法是通过StringReader类进行创建。只有字符串中每个字符的低八位可以由此类创建)。

所谓链接流处理器，就是可以接收另一个流对象作为源，并对之进行功能扩展的类。InputStream类型的链接处理器包括以下几种，它们都接收另一个InputStream对象作为流源。

- **FilterInputStream**：称为过滤输入流，它将另一个输入流作为流源。
- **ObjectInputStream**：可以将使用ObjectInputStream串行化的原始数据类型和对象重新并行化。
- **SequenceInputStream**：可以将两个已有的输入流连接起来，形成一个输入流，从而将多个输入流排列构成一个输入流序列。

注意FilterInputStream有四个子类：

- DataInputStream
- BufferedInputStream
- LineNumberInputStream
- PushbackInputStream

参照上一节的例子，可以把JDK中的原始流处理器理解为具体的Employee实现类，把FilterInputStream及其子类理解为EmployeeDecorator和其实现类，FilterInputStream类及其子类完整地实现了装饰模式。下面，来看一下JDK的实现方式和我们的例子有什么不同。

FilterInputStream的部分代码如下。

代码片段15 FilterInputStream部分代码

```
1 public class FilterInputStream extends InputStream {
2     /**
3      * 此输入对象将被增加过滤功能
4      */
5     protected InputStream in;
6     //构造方法时传入被修饰对象
7     protected FilterInputStream(InputStream in) {
8         this.in = in;
9     }
10    //其他代码
11 }
```

DataInputStream的部分实现代码如下。

代码片段16 DataInputStream部分代码

```
1 public class DataInputStream
2     extends FilterInputStream implements DataInput {
3
4     public DataInputStream(InputStream in) {
5         super(in);
6     }
7
8     public final int read(byte b[]) throws IOException {
9         return in.read(b, 0, b.length);
10    }
11
12 }
```

其UML类图如图13-6所示。

由此可见，JDK中的装饰模式与标准的装饰模式差别很小，在某些情况下，可以省略装饰抽象类直接实现装饰类，读者可以根据实际情况灵活运用。

13.2.4 装饰模式的使用范围

在以下几种情况下可以使用装饰模式：

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责时。
- 处理那些可以撤销的职责时。

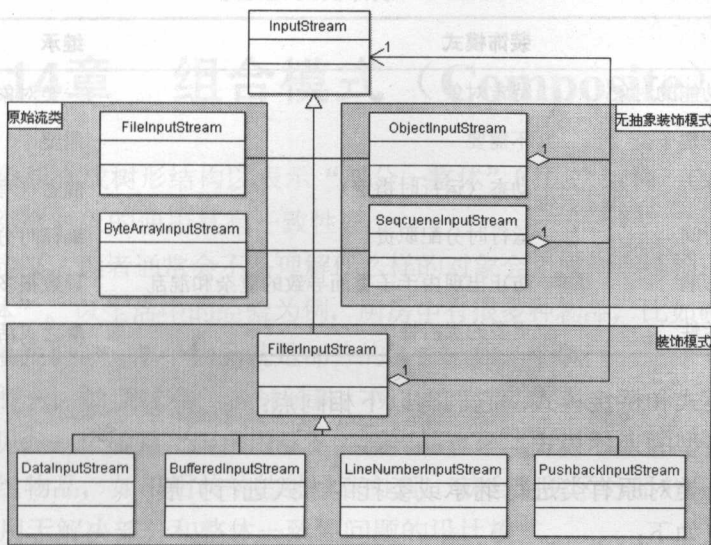


图13-6 java.io包部分类图

- 当不能采用生成子类的方法进行扩充时。这又分为两种情况，一种情况是，可能有大量独立的扩展，需要生成多种组合，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。
- 从代码角度来说，如果感觉由于功能的交叉扩展不会导致非常多的子类或者非常多的继承层次的话可以考虑装饰模式。
- 从应用角度来说，如果希望动态给类赋予或撤销一些职责，并且可以任意排列组合这些职责的话可以使用装饰模式。

13.2.5 与其他模式的关系

装饰模式是在不必改变原来的类文件和不使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰，来包裹真实的对象。

装饰模式采用对象组合而非继承的手法，实现了在运行时动态扩展对象功能的能力，而且可以根据需要扩展多个功能，避免了单独使用继承带来的“灵活性差”和“多子类衍生”问题。同时，它很好地符合面向对象设计原则中的“优先使用对象组合而非继承”和“开放—封闭”原则。

装饰模式的特点：

- 装饰对象和真实对象有相同的接口。这样客户端对象就可以用和真实对象相同的方式和装饰对象交互。
- 装饰对象包含一个真实对象的索引（reference）。
- 装饰对象接受所有的来自客户端的请求。它把这些请求转发给真实的对象。
- 装饰对象可以在转发这些请求以前或以后增加一些附加功能。这样就确保了在运行时，不用修改给定对象的结构就可以在外部增加附加的功能。

表13-1列举了装饰模式和继承方式的不同之处。

表13-1 装饰模式VS继承

	装饰模式	继承
需要扩展功能的对象	特定对象	一类对象
是否需要创建子类	不需要	需要
扩展的特征	动态（运行时指定）	静态（写在源代码中）
功能绑定时间	运行时分配职责	编译时分派职责
对子类的影响	防止出现由于子类而导致的复杂和混乱	导致很多子类产生
实现的灵活性	更多的灵活性	缺乏灵活性

对比适配器模式和桥接模式，它们有以下相同点：

- 都属于构造型的设计模式。
- 都是通过新类对原有类进行继承或委托的方式进行扩展。

它们的不同点如下：

- 对象：适配器模式强调外部接口必须一致，桥接模式强调内部实现。适配器模式通过提供新的接口形式隐蔽对原有类（功能）的调用，桥接模式把同一事物的抽象与具体行为分离。
- 封装：装饰模式可以不用修改原有对象接口，为对象增加新的功能或改变其行为；适配器模式如需增加功能，则需修改接口。

对于装饰模式与适配器模式的更多比较，可参见适配器模式章节。

思考：

- a) 装饰类所增加的功能与被装饰对象工作方法的调用顺序是否可以由客户端改变？
- b) 装饰类中可能会抛出不同的异常，怎样处理比较好？

13.3 装饰模式总结

装饰模式的作用是可以动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

在实际应用中，装饰模式通常用做给现有的类增加功能，可以根据应用的需要，对装饰模式的类进行任意顺序的包装，最后生成符合要求的对象。注意传统的装饰模式的特点是聚合了被装饰类的接口。在使用上，如果每种装饰类可能抛出不同的异常，则要对异常进行特别处理。

由此可见，JDK中的装饰模式与标准的装饰模式略有不同，它要求装饰类必须实现被装饰类的接口。读者可以根据实际情况，选择是否实现接口。在以下几种情况下可以使用装饰模式：

- 在不影响其他对象的情况下，以动态、灵活的方式对对象的功能进行扩充。
- 处理一个对象集合中的某些对象时，需要动态地改变该对象的行为。

第14章 组合模式 (Composite)

组合模式将对象组合成树形结构以表示“部分—整体”的层次结构。Composite模式使得用户对单个对象和组合对象的使用具有一致性¹。

初看此模式的定义，读者通常会不易理解什么样的对象会组成树形结构，以及什么是“部分”，什么是“整体”。以生活中的经验为例，厨房中有很多物品，比如碗、筷子、盘子、盐、酱油、蒸锅、微波炉等。单个物品就是部分的概念，如一个碗、一瓶醋等，而整体就是表示一组组合对象的概念，如调味品、各种汤碗、所有的锅等。

由以上描述可见，一个碗是“厨房用品”，调味品也是“厨房用品”，但是前者表示单个物品，后者表示一组物品，如果用传统的继承来处理，就会无法区分部分和整体。

组合模式就是用于解决部分和整体一致性问题的设计模式。

14.1 如何描述超市的组织结构

现在需要使用软件实现超市的组织结构，超市的组织结构是分层次结构的，为了直观，下面用图来说明其层次关系，如图14-1所示。

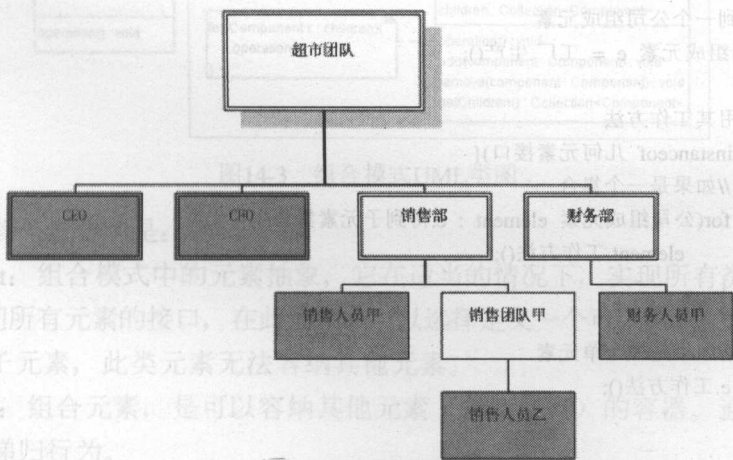


图14-1 超市组织结构图

由图14-1可见，在超市中，既有表示一个对象集合的对象，也有只表示一个独立对象的对象，而这两者可以以树型结构来表示。图14-1中深色背景结构表示独立对象，白色背景表示集合对象。

分析需求后可知，个体对象如CEO、销售人员甲等和集合对象如超市团队、财务部等都是组织结构的组成部分，但是很显然二者的操作方式不一样。对于集合对象就需要用操作集合的方法，这种集合对象不仅本身是对象，而且还可以容纳其他对象，这样客户端代码就必须区分

¹Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.GOF[95]

出对象是哪一种对象，以做出不同的操作。

初步设计的架构图如图14-2所示。

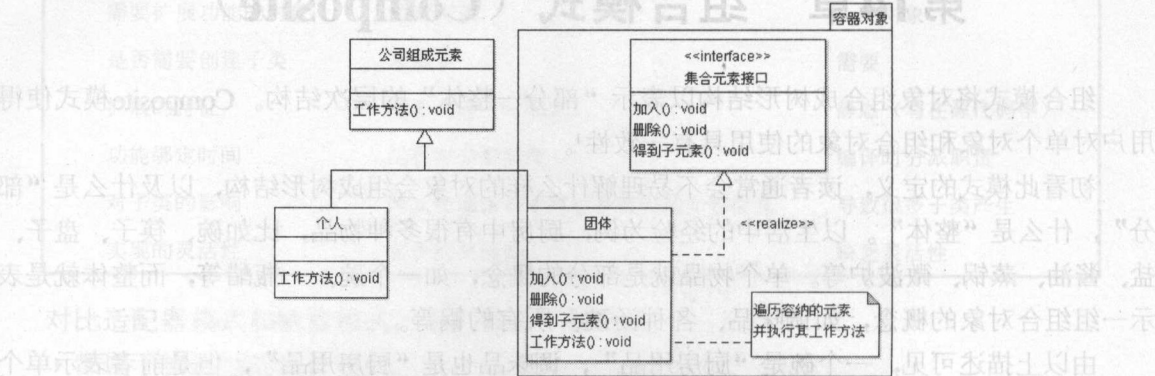


图14-2 超市组织结构的初步设计

图14-2是使用面向对象思想设计出的架构，个人和团体都是公司的组成元素，所以这是一种抽象，团体还具有集合元素接口的能力，于是又有一种接口的抽象。

这种设计看起来已经对抽象做了很好的设计，但是客户代码如果需要操作一个“公司组成元素”对象时，就必须判断是哪一种对象。其伪代码如下：

代码片段1 初步设计伪代码

```

1 //得到一个公司组成元素
2 公司组成元素 e = 工厂.生产();
3
4 //调用其工作方法
5 if(e instanceof 几何元素接口){
6     //如果是一个集合
7     for(公司组成元素 element : e.得到子元素集合()){
8         element.工作方法();
9     }
10 }else{
11     //如果是一个单元素
12     e.工作方法();
13 }
```

由代码片段1可见，客户使用时是很复杂的，因为要使用代码判断元素是独立元素还是集合元素，才能采用相应的应用代码。独立元素和集合元素的访问接口不具备一致性，增加了客户代码的使用复杂度，客户端有可能还需要进行复杂的递归编程。

思考：

- a) 代码片段1中是否可能产生递归？
- b) 客户端代码复杂度增加有何缺点？

组合模式就是用于解决对象访问接口不一致的问题的模式。

'<http://zh.wikipedia.org/zh-cn/%E9%80%92%E5%BD%92>.

14.2 组合模式的结构

组合模式的要点是如何设计单个对象和组合对象，使之具有一致的操作接口。

14.2.1 组合模式

组合模式有时候又叫做部分—整体模式，它使我们在树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

图14-3说明了此模式的结构。

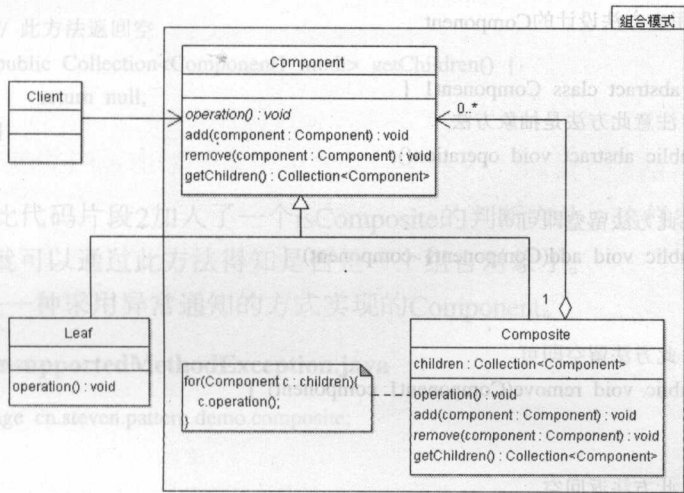


图14-3 组合模式UML类图

图14-3中的参与者分别是：

- **Component**：组合模式中的元素抽象，它在适当的情况下，实现所有类共有的方法的默认为，它是访问所有元素的接口，在此抽象中可以选择定义一个可以得到父容器元素的方法。
- **Leaf**：叶子元素，此类元素无法容纳其他元素。
- **Composite**：组合元素，是可以容纳其他元素（包括自身）的容器。通常在运行时会在这种对象上产生递归行为。
- **Client**：客户端代码，使用**Component**来操作所有的组合部件对象**Leaf**和**Composite**。

Component作为组合模式的顶级抽象，它的设计之一是要最大化接口的设计，也就是需要把**Leaf**类和**Composite**类使用到的功能全部包括。可以看到**Component**中定义的方法**add**、**remove**并不是**Leaf**所需要的方法。**Component**的设计要点是需要给某些并不是所有类都用到的方法设计一个默认的实现。这样设计的优缺点分析如下。

优点：

- 用户接口单一（对比图14-2，接口的确是单一了）。
- 默认的方法实现可以被替换（重写多态）。

缺点：

- 用户不知道所用的类是**Leaf**还是**Composite**。

• 不符合类层次结构设计原则中的规定：一个类只能定义那些对它的子类有意义的操作。
很显然，add、remove对于Leaf子类并无意义。

经过分析，图14-2和图14-3的设计中的优缺点实际上刚好是相反的。为了做出合理的设计，有时并不能面面俱到，需要做出权衡。

对于Component的设计，有以下几种设计方式，我们来分别看一下。

代码片段2 Component1.java

```
1 package cn.steven.pattern.demo.composite;
2
3 import java.util.Collection;
4
5 /**
6  * 采用空方法设计的Component
7  */
8 public abstract class Component1 {
9     // 注意此方法是抽象方法
10     public abstract void operation();
11
12     // 此方法留空即可
13     public void add(Component1 component) {
14     }
15
16     // 此方法留空即可
17     public void remove(Component1 component) {
18     }
19
20     // 此方法返回空
21     public Collection<Component1> getChildren() {
22         return null;
23     }
24 }
```

代码片段2展示了一种编程方式，其特点是操作容纳的子元素的两个方法被设计为空方法。获得子元素集合的方法被设计为直接返回空。

此方法的优点是设计简单，清晰，但是客户端进行代码调用时有可能对返回的空结果产生疑惑。对此方法进行小的改动，则得到如下设计。

代码片段3 Component1_modi.java

```
1 package cn.steven.pattern.demo.composite;
2
3 import java.util.Collection;
4
5 /**
6  * 采用空方法设计的Component修改版
7  */
8 public abstract class Component1_modi {
9     // 注意此方法是抽象方法
10     public abstract void operation();
11 }
```

```
12 // 此方法判断是否是组合对象
13 public boolean isComposite() {
14     // 默认不是组合对象
15     return false;
16 }
17
18 // 此方法留空即可
19 public void add(Component1_modi component) {
20 }
21
22 // 此方法留空即可
23 public void remove(Component1_modi component) {
24 }
25
26 // 此方法返回空
27 public Collection<Component1_modi> getChildren() {
28     return null;
29 }
30 }
```

代码片段3对比代码片段2加入了一个isComposite的判断方法，这样客户端代码得到一个Component对象时就可以通过此方法得知是否是一个组合对象了。

下面展示的是一种采用异常通知的方式实现的Component。

代码片段4 UnsupportedOperationException.java

```
1 package cn.steven.pattern.demo.composite;
2
3 /**
4  * 自定义不支持此方法的异常
5  */
6 public class UnsupportedOperationException extends Exception {
7     public UnsupportedOperationException() {
8         super("非组合对象不支持此方法");
9     }
10 }
```

代码片段5 Component2.java

```
1 package cn.steven.pattern.demo.composite;
2
3 import java.util.Collection;
4
5 /**
6  * 采用异常设计的Component
7  */
8 public abstract class Component2 {
9     // 注意此方法是抽象方法
10     public abstract void operation();
11
12     // 此方法抛出异常
13     public void add(Component2 component)
```



```

14     throws UnsupportedOperationException {
15     throw new UnsupportedOperationException();
16     }
17
18     // 此方法抛出异常
19     public void remove(Component2 component)
20     throws UnsupportedOperationException {
21     throw new UnsupportedOperationException();
22     }
23
24     // 此方法抛出异常
25     public Collection<Component2> getChildren()
26     throws UnsupportedOperationException {
27     throw new UnsupportedOperationException();
28     }
29 }

```

使用异常通知方式首先要自定义一个异常，如代码片段4所示。

代码片段5修改了代码片段2的一个缺点，现在客户端在调用Leaf上的集合操作方法时就会得到一个异常的通知。在Java体系中，这样的设计是官方推荐的，它具有设计的完备性。

如下Component的设计是借鉴了.NET Framework¹中的组合模式代码，采用直接返回集合对象的方式操作集合。

代码片段6 Component3.java

```

1 package cn.steven.pattern.demo.composite;
2
3 import java.util.Collection;
4
5 /**
6  * 采用集合设计的Component
7  */
8 public abstract class Component3 {
9     // 注意此方法是抽象方法
10    public abstract void operation();
11
12    // 此方法抛出异常
13    public Collection<Component3> getChildren()
14    throws UnsupportedOperationException {
15    throw new UnsupportedOperationException();
16    }
17 }

```

代码片段6展示了一种更简单的编程方式，代码中并没有定义对集合增删的方法，而是直接给客户端一个集合来进行操作，这归功于在Java中Collection及其子接口定义了完备的操作方法，所以不需要我们自己实现操作方法了，直接使用JDK中的集合操作功能即可。

通过以上各例可以看出，有多种方式实现Component类，它们有各自的优缺点，在下面的

¹http://zh.wikipedia.org/zh-cn/.NET_Framework.

代码中, 将采用代码片段5所使用的设计方法。

如下展示的是Composite组合类的代码, 请注意在代码中对集合的处理方法。

代码片段7 Composite.java

```

1 package cn.steven.pattern.demo.composite;
2
3 import java.util.Collection;
4 import java.util.LinkedList;
5
6 /**
7  * 组合类
8  */
9 public class Composite extends Component2 {
10
11     // 私有的集合属性
12     private Collection<Component2> children;
13
14     // 构造方法
15     public Composite() {
16         super();
17         // 构造时创建集合容器
18         children = new LinkedList<Component2>();
19     }
20
21     //父类的抽象方法必须重写
22     @Override
23     public void operation() {
24         System.out.println("Composite operation");
25         // 遍历操作容器中的对象
26         try {
27             for (Component2 component : getChildren()) {
28                 // 此处产生递归
29                 component.operation();
30             }
31         } catch (UnsupportedMethodException e) {
32             e.printStackTrace();
33         }
34     }
35
36     // 必须重写add方法
37     @Override
38     public void add(Component2 component)
39         throws UnsupportedOperationException {
40         getChildren().add(component);
41     }
42
43     // 必须重写remove方法
44     @Override
45     public void remove(Component2 component)
46         throws UnsupportedOperationException {
47         getChildren().remove(component);

```

```
48     }
49
50     // 必须重写getChildren方法
51     @Override
52     public Collection<Component2> getChildren()
53         throws UnsupportedOperationException {
54         return children;
55     }
56
57     public void setChildren(Collection<Component2> children) {
58         this.children = children;
59     }
60 }
```

代码片段7展示了组合模式中组合类的代码，注意其中必须重写父类关于集合操作的各个方法，这样就会在运行时刻采用多态的形式实现集合功能。

下面展示的是叶子元素的代码。

代码片段8 Leaf.java

```
1 package cn.steven.pattern.demo.composite;
2
3 /**
4  * 单一叶子类
5  */
6 public class Leaf extends Component2 {
7
8     // 父类的抽象方法必须重写
9     @Override
10    public void operation() {
11        System.out.println("Leaf operation");
12    }
13
14 }
```

代码片段8的代码非常简练，只实现了一个父类的抽象方法，其他的集合操作方法将从父类中继承得来，这种设计的Leaf代码量是很少的。

下面通过客户端代码实现如图14-4所示的一个组合结构，通过操作可以看出组合模式的工作方法。

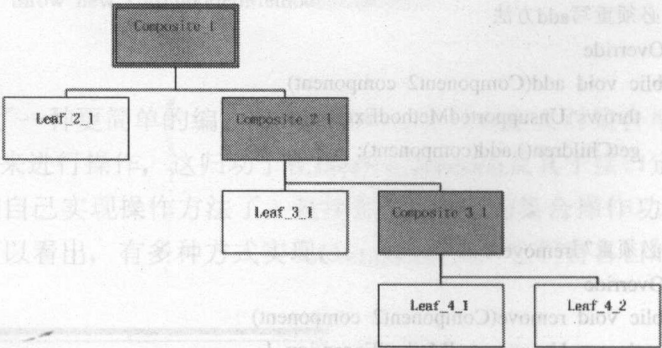


图14-4 组合结构图

注意图14-4中深色背景的是组合对象，白色背景的是单一对象。下面是客户端代码。

代码片段9 Client.java

```
1 package cn.steven.pattern.demo.composite;
2
3 /**
4  * 组合模式使用客户端
5  */
6 public class Client {
7     public static void main(String[] args) {
8         // 创建组合对象
9         Composite composite_1 = new Composite();
10        Composite composite_2_1 = new Composite();
11        Composite composite_3_1 = new Composite();
12
13        // 创建单一对象
14        Leaf leaf_2_1 = new Leaf();
15        Leaf leaf_3_1 = new Leaf();
16        Leaf leaf_4_1 = new Leaf();
17        Leaf leaf_4_2 = new Leaf();
18
19        // 从最下层开始装配
20        try {
21            composite_3_1.add(leaf_4_1);
22        } catch (UnsupportedMethodException e) {
23        }
24        try {
25            composite_3_1.add(leaf_4_2);
26        } catch (UnsupportedMethodException e) {
27            e.printStackTrace();
28        }
29
30        try {
31            composite_2_1.add(leaf_3_1);
32        } catch (UnsupportedMethodException e) {
33        }
34        try {
35            composite_2_1.add(composite_3_1);
36        } catch (UnsupportedMethodException e) {
37            e.printStackTrace();
38        }
39
40        try {
41            composite_1.add(leaf_2_1);
42        } catch (UnsupportedMethodException e) {
43            e.printStackTrace();
44        }
45        try {
46            composite_1.add(composite_2_1);
47        } catch (UnsupportedMethodException e) {
```

```
49         e.printStackTrace();
50     }
51
52     // 下面就可以使用树形结构中的任意一个节点了
53     System.out.println("tLOG:composite_1.operation()");
54     composite_1.operation();
55     System.out.println("tLOG:leaf_3_1.operation()");
56     leaf_3_1.operation();
57     System.out.println("tLOG:composite_3_1.operation()");
58     composite_3_1.operation();
59 }
60
61 }
```

在代码片段9第9行~第17行创建各种对象时顺序并不重要，第20行~第50行装配组合对象时要注意逻辑顺序，应从上至下装配（从上至下也可以），如果不按照顺序的话容易乱了逻辑。由代码第54行、第56行、第58行可见，客户端代码对于两种对象的使用代码是一致的。

图14-5所示的是执行结果。

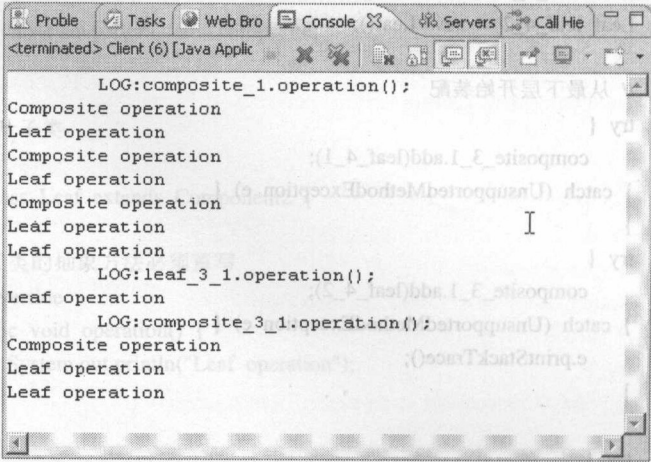


图14-5 组合模式的执行结果

由图14-5可见，客户端对组合对象和单一对象的操作是一样的，但是执行结果不一样，单一对象直接调用operation方法，组合对象采用递归的方式调用集合中的每一个子对象。组合模式帮助我们为这种结构解决了一致性访问的问题。

14.2.2 使用桥接模式解决超市组织结构问题

经过了上一节的学习，下面我们将应用组合模式来解决超市组织结构的问题。

由图14-1可知超市的组织结构可以归结为“部分-整体”的问题，单个个人职位可以看做是一个个体，它不具备容器的特性，而一个部门是一个组合结构，它不光可以容纳个体，还可以容纳部门这种组合。

参照图14-3，可以有以下的设计，注意其中的功能有部分扩展，如图14-6所示。

由于篇幅有限，图14-6中并没有把类中定义的所有方法列出，只列出了有代表性的几个方法，具体代码参见后面的代码片段部分。

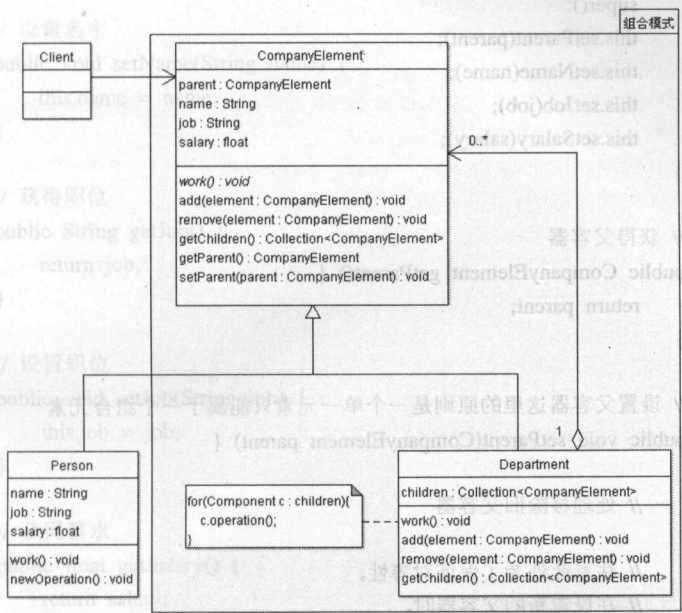


图14-6 使用组合模式解决超市组织结构问题

首先看到的是最高层抽象CompanyElement的代码，注意此代码比较复杂，比原始的组合模式增加了功能。

代码片段10 CompanyElement.java

```
1 package cn.steven.pattern.demo.composite.example;
2
3 import java.util.Collection;
4
5 /**
6  * 组合模式高层抽象
7  */
8 public abstract class CompanyElement {
9     // 父容器
10    private CompanyElement parent;
11    // 个人名或部门名
12    private String name;
13    // 职位或部门工作类属
14    private String job;
15    // 工资，个人工资或部门工资和
16    private float salary;
17
18    // 空构造方法
19    public CompanyElement() {
20    }
21
22    // 带参数构造方法
23    public CompanyElement(CompanyElement parent,
24        String name, String job,
25        float salary) {
```



```

26     super();
27     this.setParent(parent);
28     this.setName(name);
29     this.setJob(job);
30     this.setSalary(salary);
31 }
32
33 // 获得父容器
34 public CompanyElement getParent() {
35     return parent;
36 }
37
38 // 设置父容器这里的原则是一个单一元素只能属于一个组合元素
39 public void setParent(CompanyElement parent) {
40
41     // 处理移除旧父容器
42
43     // 注意此处为了保证对等性,
44     // 在设置新的父容器时,
45     // 必须移除以前的父容器
46     if (getParent() != null && getParent() != parent) {
47         try {
48             if (getParent().getChildren().contains(this)) {
49                 getParent().remove(this);
50             }
51         } catch (UnsupportedMethodException e) {
52             e.printStackTrace();
53         }
54     }
55
56     // 处理设置父容器
57     try {
58         // 注意此处为了保证对等性,
59         // 在设置父容器时,
60         // 必须在容器中也加入此对象
61         this.parent = parent;
62
63         if (parent != null &&
64             !parent.getChildren().contains(this)) {
65             parent.getChildren().add(this);
66         }
67     } catch (UnsupportedMethodException e) {
68         e.printStackTrace();
69     }
70 }
71
72
73 // 获得名字
74 public String getName() {
75     return name;
76 }

```

```

77 // 设置名字
78 public void setName(String name) {
79     this.name = name;
80 }
81
82 // 获得职位
83 public String getJob() {
84     return job;
85 }
86
87 // 设置职位
88 public void setJob(String job) {
89     this.job = job;
90 }
91
92 // 获得薪水
93 public float getSalary() {
94     return salary;
95 }
96
97 // 设置薪水 (在组合中将被重写)
98 public void setSalary(float salary) {
99     this.salary = salary;
100 }
101
102 // 注意此方法是抽象方法
103 public abstract void work();
104
105 // 此方法抛出异常
106 public void add(CompanyElement component)
107     throws UnsupportedOperationException {
108     throw new UnsupportedOperationException();
109 }
110
111 // 此方法抛出异常
112 public void remove(CompanyElement component)
113     throws UnsupportedOperationException {
114     throw new UnsupportedOperationException();
115 }
116
117 // 此方法抛出异常
118 public Collection<CompanyElement> getChildren()
119     throws UnsupportedOperationException {
120     throw new UnsupportedOperationException();
121 }
122
123 // 此方法抛出异常
124 public void setChildren(Collection<CompanyElement> children)
125     throws UnsupportedOperationException {
126     throw new UnsupportedOperationException();
127 }

```

```

128     }
129     // 设置父容器
130     // 重写描述方法
131     @Override
132     public String toString() {
133         StringBuffer sb = new StringBuffer();
134         // 使用递归方式处理此元素的层次描述
135         if (this.getParent() != null) {
136             // 如果有父容器
137             sb.append(this.getParent().toString() + " 的下属 ");
138         } else {
139             // 如果没有父容器不写入信息
140         }
141         sb.append(this.getName() + "[" + this.getJob() + "]");
142         return sb.toString();
143     }
144 }

```

注意代码片段10中第38行~第71行是设置父容器的方法，此方法为新增加的功能，它的作用是可以对元素设置父容器来使之加入到父容器中，并且由于parent属性的引入，任何一个元素都可以得到它的父容器，这样就有了反向遍历的功能了。

注意代码片段10中第130行~第143行重写了toString方法，此方法可以供客户端以简单的方式得到元素的文字描述信息，此处由于使用了parent的调用，所以隐含了递归的过程。

UnsupportedMethodException.java代码参见代码片段4。

下面展示的是单个元素的代码。

代码片段11 Person.java

```

1  package cn.steven.pattern.demo.composite.example;
2
3  public class Person extends CompanyElement {
4
5      // 重写抽象方法
6      @Override
7      public void work() {
8          System.out.println(this.toString() + " 工作了!");
9      }
10
11     // 空参数构造方法
12     public Person() {
13         super();
14     }
15
16     // 带参数构造方法
17     public Person(CompanyElement parent,
18         String name, String job, float salary) {
19         super(parent, name, job, salary);
20     }
21 }

```

代码片段11十分简单，需要注意的是第8行中的toString方法的调用中隐含的递归过程。构

造方法第17行~第20行是为了方便客户程序使用而设计的。

下面展示的是组合元素的代码。

代码片段12 Department.java

```
1 package cn.steven.pattern.demo.composite.example;
2
3 import java.util.Collection;
4 import java.util.LinkedList;
5
6 public class Department extends CompanyElement {
7
8     private Collection<CompanyElement> children
9         = new LinkedList<CompanyElement>();
10
11     // 重写容器方法
12     @Override
13     public Collection<CompanyElement> getChildren()
14         throws UnsupportedOperationException {
15         return children;
16     }
17
18     // 重写容器方法
19     @Override
20     public void setChildren(
21         Collection<CompanyElement> children)
22         throws UnsupportedOperationException {
23         this.children = children;
24     }
25
26     // 重写抽象方法
27     @Override
28     public void work() {
29         System.out.println(this.toString() + " 通知下属！");
30         try {
31             // 通知容器中的单个元素执行工作方法
32             for (CompanyElement e : getChildren()) {
33                 e.work();
34             }
35         } catch (UnsupportedOperationException e) {
36             e.printStackTrace();
37         }
38     }
39
40     // 重写获得薪水方法
41     @Override
42     public float getSalary() {
43         // 初始化集合的薪水
44         this.setSalary(0f);
45         // 遍历子元素将薪水相加
46         for (CompanyElement e : children) {
47             this.setSalary(super.getSalary() + e.getSalary());
```

```

48     }
49     // 返回加完后的薪水
50     return super.getSalary();
51 }
52
53 // 空参数构造方法
54 public Department() {
55     super();
56 }
57
58 // 带参数构造方法
59 public Department(CompanyElement parent,
60     String name, String job,
61     float salary) {
62     super(parent, name, job, salary);
63 }
64
65 // 重写集合操作方法
66 @Override
67 public void add(CompanyElement component)
68     throws UnsupportedOperationException {
69     // 设计setParent时已考虑到对等性问题
70     component.setParent(this);
71 }
72
73 // 重写集合操作方法
74 @Override
75 public void remove(CompanyElement component)
76     throws UnsupportedOperationException {
77     // 设计setParent时已考虑到对等性问题
78     component.setParent(null);
79 }
80 }

```

代码片段12中第8行~第9行定义了所要使用的集合容器，重写了getChildren、setChildren、add、remove、getSalary方法，注意其中第40行~第51行是怎样获得集合中每一个元素的薪水的，其中隐含了递归的过程。

注意代码片段12中第65行~第79行操作集合的方式，因为在CompanyElement中的setParent方法已经设计得很完备了，所以在这里调用这个方法即可，如果直接操作集合，还会存在对等性的问题。

最后展示的是客户端代码。

代码片段13 Client.java

```

1 package cn.steven.pattern.demo.composite.example;
2
3 /**
4  * 超市组织结构客户端
5  */
6 public class Client {

```

```

7
8 public static void main(String[] args) {
9
10 // 实例化超市组织的对象
11 public CompanyElement root = new Department(
12     null, "超市团队", "", 0f);
13 CompanyElement saleDept = new Department(
14     root, "销售部", "销售", 0f);
15 CompanyElement saleDept_1 = new Department(
16     saleDept, "销售团队甲", "销售", 0f);
17 CompanyElement financialDept = new Department(
18     root, "财务部", "后勤", 0f);
19
20 CompanyElement ceo = new Person(
21     root, "岳不群", "CEO", 10000f);
22 CompanyElement cfo = new Person(
23     root, "宁中则", "CFO", 8000f);
24 CompanyElement saler_1 = new Person(
25     saleDept, "令狐冲", "销售人员", 3000f);
26 CompanyElement saler_2 = new Person(
27     saleDept_1, "陆大有", "销售人员", 2000f);
28 CompanyElement financial_1 = new Person(
29     financialDept, "岳灵珊", "财务人员", 2000f);
30
31 System.out.println("测试根:");
32 root.work();
33 System.out.println("测试组合对象:");
34 saleDept.work();
35 System.out.println("测试单个对象:");
36 saler_2.work();
37
38 System.out.println(root.getName() + "的工资是:"
39     + root.getSalary());
40 System.out.println(cfo.getName() + "的工资是:"
41     + cfo.getSalary());
42 }
43
44 }

```

注意代码片段13中第32行~第36行对于不同对象使用了相同的**Work**方法，第38行~第41行对不同对象获得的薪水也使用了相同的方法，而对于客户端的操作也是一致的。

代码执行的结果如图14-7所示。

图14-7可见，经过使用组合模式处理超市组织结构，终于解决了对于“部分”和“整体”的操作一致性的问题，而且还获得了正向和反向查询的功能。

现在，增加一个单一对象或组合对象也是很简单的，只需要调用某个组合对象的**add**或本对象的**setParent**方法即可。移除对象的话使用组合对象的**remove**或本对象的**setParent**方法即可。

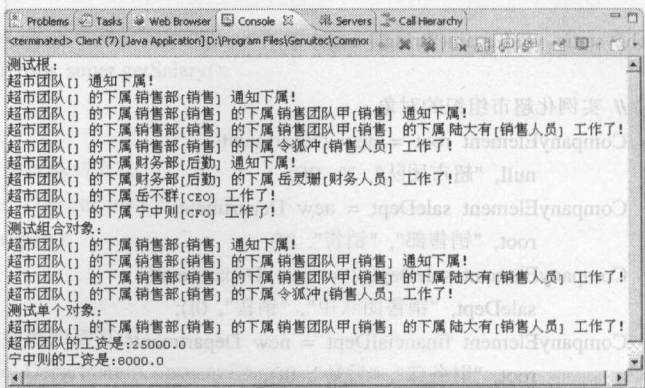


图14-7 Client执行结果

14.2.3 组合模式在JDK中的实例

Composite模式在GOF定义的机构上使用时有一点局限性，无法直接看出是Leaf节点还是Composite节点，有时会对客户端造成一定的困扰。

JDK的AWT设计中就采用了一种更改过的组合模式，类图如图14-8所示。

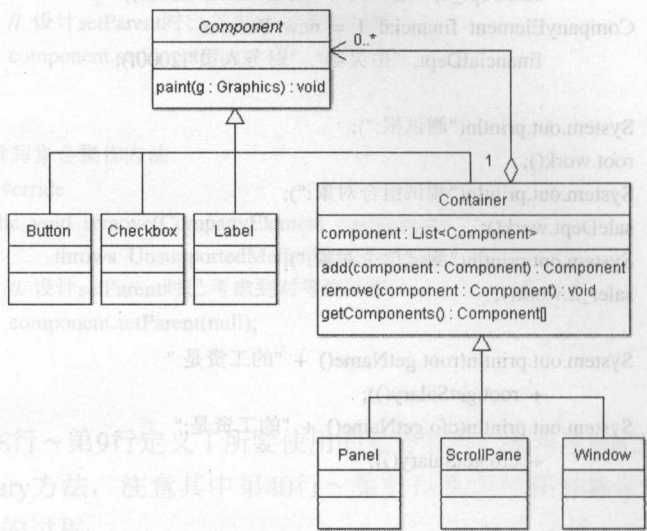


图14-8 JDK中AWT包的组合模式设计

注意图14-8与图14-3的设计的不同之处在于Component中是否设计了集合的操作方法，如add、remove等，这是一个设计的“透明性”和“安全性”的问题。

GOF组合模式是将集合操作方法设计在Component中的，这对于客户是一个很方便的设计，保证了接口的“透明性”，但是可能会产生“安全性”的问题，客户可能会在单一对象上执行集合操作方法。

JDK中的设计是不将集合操作方法设计在Component中，而是设计在一个Container类中，所有容器类都继承这个类。很显然，客户如果想使用集合操作，必须明确知道对象是一个Container，这样客户的代码就复杂了一些，顶级接口也失去了“透明性”，但是“安全性”的问题就没有了，客户不会再操作一个不存在的方法。

下面看一下两个主要类的部分代码:

代码片段14 Component.java部分代码

```

1 public abstract class Component implements ImageObserver,
2     MenuContainer, Serializable {
3     /**
4      * 桥接模式的对等对象
5      */
6     transient ComponentPeer peer;
7
8     /**
9      * 此元素的父容器
10    */
11    transient Container parent;
12
13    /**
14     * 图形元素的坐标和尺寸
15     */
16    int x;
17    int y;
18    int width;
19    int height;
20
21    /**
22     * 元素名称
23     */
24    public String getName() {
25        if (name == null && !nameExplicitlySet) {
26            synchronized (getObjectLock()) {
27                if (name == null && !nameExplicitlySet)
28                    name = constructComponentName();
29            }
30        }
31        return name;
32    }
33
34    public void setName(String name) {
35        String oldName;
36        synchronized (getObjectLock()) {
37            oldName = this.name;
38            this.name = name;
39            nameExplicitlySet = true;
40        }
41        firePropertyChange("name", oldName, name);
42    }
43
44    /**
45     * 画出此元素
46     */
47    public void paint(Graphics g) {

```

```
48     }  
49 }
```

由代码片段14的部分代码可见，在Component中并没有设计集合操作的方法，此类是一个组合模式和桥接模式的综合应用。注意代码第11行定义了一个表示父容器的属性，这样就可以由一个对象得到其父对象，进而回溯至根元素。

代码片段15 Container.java部分代码

```
1  public class Container extends Component {  
2  
3      /**  
4       * 元素的容器  
5       */  
6      private java.util.List<Component> component  
7          = new java.util.ArrayList<Component>();  
8  
9      /**  
10     * 布局管理器  
11     */  
12     LayoutManager layoutMgr;  
13  
14     /**  
15     * 轻量级组件事件分发器  
16     */  
17     private LightweightDispatcher dispatcher;  
18  
19     /**  
20     * 得到所有子组件  
21     */  
22     public Component[] getComponents() {  
23         return getComponents_NoClientCode();  
24     }  
25  
26     /**  
27     * 得到子组件的具体方法  
28     */  
29     final Component[] getComponents_NoClientCode() {  
30         return component.toArray(EMPTY_ARRAY);  
31     }  
32  
33     /**  
34     * 新增一个组件  
35     */  
36     public Component add(Component comp) {  
37         addImpl(comp, null, -1);  
38         return comp;  
39     }  
40  
41     /**  
42     * 重载新增组件  
43     */
```



```
44 public Component add(String name, Component comp) {
45     addImpl(comp, name, -1);
46     return comp;
47 }
48
49 /**
50  * 重载新增组件
51  */
52 public Component add(Component comp, int index) {
53     addImpl(comp, null, index);
54     return comp;
55 }
56
57 /**
58  * 加入子元素的具体实现方法
59  */
60 protected void addImpl(Component comp,
61     Object constraints, int index) {
62     //代码略
63 }
64
65 /**
66  * 移除子元素
67  */
68 public void remove(int index) {
69     //代码略
70 }
71
72 /**
73  * 重载移除子元素
74  */
75 public void remove(Component comp) {
76     synchronized (getTreeLock()) {
77         if (comp.parent == this) {
78             int index = component.indexOf(comp);
79             if (index >= 0) {
80                 remove(index);
81             }
82         }
83     }
84 }
85
86 /**
87  * 移除所有元素
88  */
89 public void removeAll() {
90     // 代码略
91 }
92 }
```

由代码片段15的部分代码可见, 其中设计了容器的相关操作方法, 这样客户在使用一个组

合对象时必须使用Container来描述，而不能使用Component了。

由上例可见，JDK中为了“安全性”的原因对GOF的组合模式做了改进，读者在使用此模式时可以根据自己的需要进行变化，不必局限于GOF指定的方式。

14.2.4 组合模式的使用范围

组合模式的使用范围如下：

- 多个对象之间存在层次组合的结构。
- 要使组合对象或者单个对象对于客户而言具有访问的一致性。

组合模式的优点有：

- 客户端操作对象很容易，各种对象的操作具有一致性。
- 增加各种类型的子类都很容易。

组合模式的缺点有：

- 有安全性问题，客户的方法调用可能不安全。

14.2.5 与其他模式的关系

组合模式可以用于命令的组成，例如由一些相互关联的命令组成一个复杂的批处理命令。在使用命令模式时可以使用这种模式。

在使用Composite模式的同时，可以使用装饰模式进行功能的扩展，但要注意装饰模式类和Composite类具有相同的父类，并实现各种集合操作方法。

在访问Composite对象时，推荐使用Iterator模式来抽象获得元素的方法。如果直接使用集合操作的话容易使访问的具体方法和客户代码耦合度过高。

14.3 组合模式总结

Composite模式采用树形结构来实现普遍存在的对象容器，从而将“一对多”的关系转化为“一对一”的关系，使得客户代码可以一致地处理对象和对象容器，无需关心处理的是单个对象，还是组合的对象容器。

“将客户代码与复杂的对象容器结构解耦”是Composite模式的核心思想，解耦之后，客户代码将与纯粹的对象接口（而非对象容器的复杂内部实现结构）发生依赖关系，从而更能“应对变化”。

Composite模式在具体实现中，可以让父对象中的子对象反向追溯：如果父对象有频繁的遍历需求，可使用缓存技巧来改善效率。

Composite模式中，是将“Add和Remove的与对象容器相关的方法”定义在“表示抽象对象的Component类”中，还是将其定义在“表示对象容器的Composite类”中，是一个关乎“透明性”和“安全性”的两难问题，需要仔细权衡，而且这又是必须付出的代价。GOF定义的组合模式是将集合操作方法定义在Component类中，而JDK中是定义在Composite类中的。

当需求中存在“部分-整体”问题时，需要使用组合模式解决用户接口一致性的问题。

第15章 享元模式 (Flyweight)

享元模式的意图是用共享技术有效地支持大量的细粒度对象¹。

面向对象的思想确实很好地解决了抽象性的问题，以至于在面向对象设计者的眼中，万事万物一切皆对象。但不可避免的是，采用面向对象的编程方式可能会增加一些资源和性能上的开销。不过，在大多数情况下，这种影响还不是太大，所以，它带来的空间和性能上的损耗相对于它的优点而言，基本上不用考虑。但是，在某些特殊情况下，大量细粒度对象的创建、销毁以及存储所造成的资源和性能上的损耗，可能会在系统运行时形成瓶颈。那么我们该如何去避免产生大量的细粒度对象，同时又不影响系统使用面向对象的方式进行操作呢？享元设计模式提供了一个比较好的解决方案。

公共电话网的使用方式就是生活中常见的享元模式的例子。公共电话网中的一些资源，例如拨号音发生器、振铃发生器和拨号接收器，都是必须由所有用户共享的，不可能为每一个人都配备一套这样的资源，否则公共电话网的资源开销也太大了。当一个用户拿起听筒打电话时，他根本不需要知道到底使用了多少资源，对用户而言所有的事情就只有拨号音，拨打号码，拨通电话就行了。所以，就有很多人会共用一套资源，这就是享元模式的基本思想。

15.1 宣传海报设计过程中的思考

超市需要设计相当多的海报配合广告宣传，设计部的员工为了满足各种需求往往要设计各种不同风格的海报，这些海报有的以文字为主，如图15-1所示。还有的以图形为主，如图15-2所示。



图15-1 文字海报

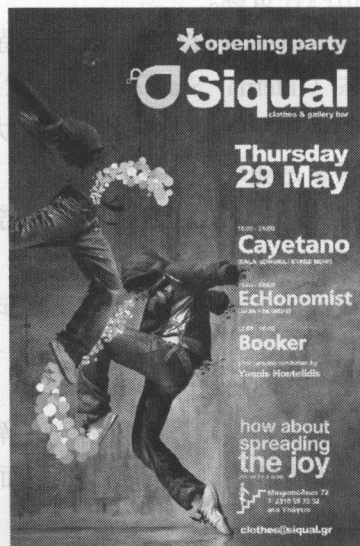


图15-2 图文海报

¹Use sharing to support large numbers of fine-grained objects efficiently. GOF[95]

海报设计人员需要和程序员配合以实现其创意，海报的设计中充斥着大量的文字和图片，每一个文字和图片在面向对象的设计中都应该当做一个对象，这样就会导致一个严重的问题：对象数量爆炸。

每一个字符或图片对象至少有下列几项属性：

- 文字内码。
- 位置。
- 颜色。
- 字体。
- 大小。

经过考察，会发现相同的字母（以英文为例）有时会在一张海报上出现多次，有可能每次出现时的大小、位置、字体和颜色都不同。

按照面向对象的思想，任何一个独立的实体都可以设计为一个对象，这在通常的应用中，不会出现问题，因为一般的应用涉及到的对象的数目不多，但是设计海报则不同，因为其中充斥着大量的文字信息，英文的26个字母会以不同的形态重复若干次，每一个都会被设计成一个独立的对象。如果按照传统的面向对象思想来设计具有很多相似对象的系统，最大的问题就是由于内存的占用量巨大，使系统开销很大。

如果想解决对象过多的问题，最好的方法就是使很多对象共享信息，享元模式就是采用共享的方式减少对象数量的一种模式。

15.2 享元模式的结构

Flyweight在拳击比赛中指次最轻量级，即“蝇量级”。这里使用它更能反映享元模式的本质。在编程中，资源开销大的称之为“重量级”，不言而喻，享元模式的资源开销就很小，在这里，尤指内存资源。

享元模式以共享的方式高效地支持大量的细粒度对象。享元对象能做到共享的关键在于区分内蕴状态（Internal State）和外蕴状态（External State）。内蕴状态存储在享元对象内部并且不会随环境改变而改变，因此内蕴状态并可以共享。

外蕴状态是随环境改变而改变的、不可以共享的状态。享元对象的外蕴状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部。外蕴状态与内蕴状态是相互独立的。

享元模式的实现分为单纯享元模式和复合享元模式两种。

15.2.1 单纯享元模式

下面以为一篇文章排版为例，讲解享元模式。英文字母的大量重复是导致对象数量过多的主要原因，为了解决这个问题，在下面的设计中将保证每一种字母对象只创建一个实例，如图15-3所示。

注意图15-3展示了其基本结构，其中的参与者如下：

- CharFlyweight：此抽象类的作用是保存内蕴状态，描述外蕴状态，提供外部代码操作享元对象的方法。

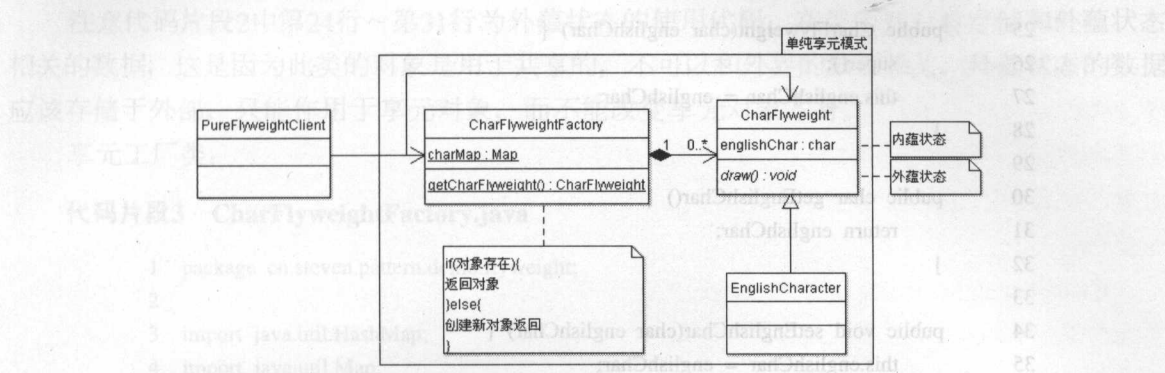


图15-3 单纯享元模式UML类图

- **EnglishCharacter**: 具体实现的享元类，提供了外蕴状态的具体操作方法，通常并不会存储具体的外蕴状态数据。
- **CharFlyweightFactory**: 管理享元对象，此类是保证享元对象被共享的关键类，通常使用对象池的方式存储享元对象。
- **PureFlyweightClient**: 操作享元对象的客户类，注意此类使用享元对象时必须通过享元工厂类，它可以计算或存储享元对象的外蕴状态。

下面展示的是具体的实现代码，代码的难点是CharFlyweightFactory中关于维护对象池的相关代码。

享元抽象类：

代码片段1 CharFlyweight.java

```
1 package cn.steven.pattern.demo.flyweight;
2
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.Point;
6
7 /**
8  * 单纯享元模式抽象类
9  */
10 public abstract class CharFlyweight {
11     /**
12      * 内蕴状态
13      */
14     private char englishChar;
15
16     /**
17      * 外蕴状态
18      * @param position 字符位置
19      * @param color 字符颜色
20      * @param size 字号大小
21      * @param font 字体
22      */
23     public abstract void draw(Point position, Color color, Font font);
24 }
```

```
25     public CharFlyweight(char englishChar) {
26         super();
27         this.englishChar = englishChar;
28     }
29
30     public char getEnglishChar() {
31         return englishChar;
32     }
33
34     public void setEnglishChar(char englishChar) {
35         this.englishChar = englishChar;
36     }
37
38 }
```

享元实现类:

代码片段2 EnglishCharacter.java

```
1  package cn.steven.pattern.demo.flyweight;
2
3  import java.awt.Color;
4  import java.awt.Font;
5  import java.awt.Point;
6
7  /**
8   * 具体字符类
9   */
10 public class EnglishCharacter extends CharFlyweight {
11
12     /**
13      * 设置内蕴状态
14      *
15      * @param c
16      */
17     public EnglishCharacter(char c) {
18         super(c);
19     }
20
21     /**
22      * 设置外蕴状态
23      */
24     @Override
25     public void draw(Point position, Color color, Font font) {
26         System.out.println("字符: " + getEnglishChar());
27         System.out.println("\t位置: X:" + position.getX() + " Y:"
28             + position.getY());
29         System.out.println("\t颜色: " + color.toString());
30         System.out.println("\t字体: " + font.toString());
31     }
32
33 }
```


注意代码片段2中第24行~第31行为外蕴状态的使用代码，在类中并没有存储和外蕴状态相关的数据，这是因为此类的对象是用于共享的，不可以和外界的数据相关，外蕴状态的数据应该存储于外部，只能作用于享元对象，而不能改变享元对象本身。

享元工厂类：

代码片段3 CharFlyweightFactory.java

```
1 package cn.steven.pattern.demo.flyweight;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * 享元模式的工厂方法
8  */
9 public class CharFlyweightFactory {
10
11     /**
12      * Flyweight对象缓存池
13      */
14     private static Map<Character, CharFlyweight> charMap =
15         new HashMap<Character, CharFlyweight>();
16
17     public synchronized static CharFlyweight getCharFlyweight(char c) {
18         /**
19          * 查看此对象是否已经存在
20          */
21         if (charMap.containsKey(c)) {
22             // 存在此对象则直接使用原有对象
23             return charMap.get(c);
24         } else {
25             // 不存在此对象生成新对象
26             CharFlyweight charFlyweight = new EnglishCharacter(c);
27             // 存入集合中
28             charMap.put(c, charFlyweight);
29             return charFlyweight;
30         }
31     }
32
33     /**
34      * 显示Flyweight对象缓存池状态
35      */
36     public static void showStatus(){
37         System.out.print("flyweight对象数量: "+charMap.size()+" 字符列表: ");
38         for (Map.Entry<Character, CharFlyweight> entry : charMap.entrySet()) {
39             System.out.print(entry.getKey()+" ");
40         }
41         System.out.println();
42     }
43 }
```

注意代码片段3中所有的方法都是静态的，`charMap`为存放享元对象的缓冲池，操作的方法为`getCharFlyweight`方法，由于存在多线程共享的问题，所以在此方法上声明了`synchronized`特性。代码第17行~第31行为此工厂类的主要代码，功能是根据要获得的字符来共享享元对象。如客户想得到一个A字符的享元对象，则此方法先在`charMap`中查找是否已经存在此字符对应的对象，如果有，就直接返回原有对象，如果没有，则生成新享元对象，将其加入`charMap`，再返回给客户。

`CharFlyweightFactory`通过这种方式来保证享元对象的唯一性。只要能保证只使用此类获得享元对象，则可以保证享元对象的共享性。

客户端代码:

代码片段4 PureFlyweightClient.java

```
1 package cn.steven.pattern.demo.flyweight;
2
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.Point;
6
7 /**
8  * 单纯享元模式测试类
9  */
10 public class PureFlyweightClient {
11     /**
12      * 显示OOP三个字符
13      */
14     public static void main(String[] args) {
15         // 显示Flyweight对象缓存池状态
16         CharFlyweightFactory.showStatus();
17         CharFlyweight char1 = CharFlyweightFactory.getCharFlyweight('O');
18         CharFlyweight char2 = CharFlyweightFactory.getCharFlyweight('O');
19         CharFlyweight char3 = CharFlyweightFactory.getCharFlyweight('P');
20         // 显示Flyweight对象缓存池状态
21         CharFlyweightFactory.showStatus();
22         // 输出字符
23         char1.draw(new Point(0, 0), Color.red, new Font("宋体", Font.BOLD, 1));
24         char2.draw(new Point(1, 1), Color.red, new Font(Font.SERIF,
25             Font.ITALIC, 5));
26         char3.draw(new Point(2, 5), Color.red, new Font(Font.MONOSPACED,
27             Font.PLAIN, 8));
28     }
29
30 }
```

运行结果如图15-4所示。

结合代码片段4和图15-4可以看出初始享元对象的个数为0，使用`CharFlyweightFactory`对应的方法请求了O、O、P三个对象后对象的个数为2，因为有两个对象其实共享了同一个对象。

单纯享元模式应对的情形比较简单，如果共享的情形比较复杂，则需要使用复合享元模式。

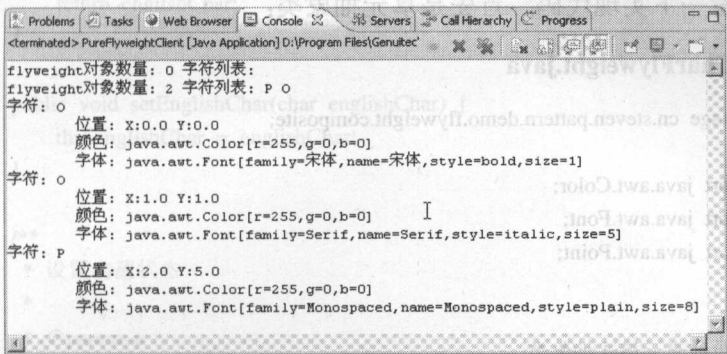


图15-4 PureFlyweightClient.java运行结果

15.2.2 复合享元模式

上一小节中所介绍的单纯享元模式其实是一种最简单的享元情形，更多的情况下，我们需要将小的享元对象组织起来统一管理，比如一个文档，最小的单位是字符，字符再组成行，行又组成段落，段落组成章节，在这样的组合中，可以看到组合模式的影子，复合享元模式就是采用组合模式的思想 and 享元模式共同组成的组织架构方法。其典型类图如图15-5所示。

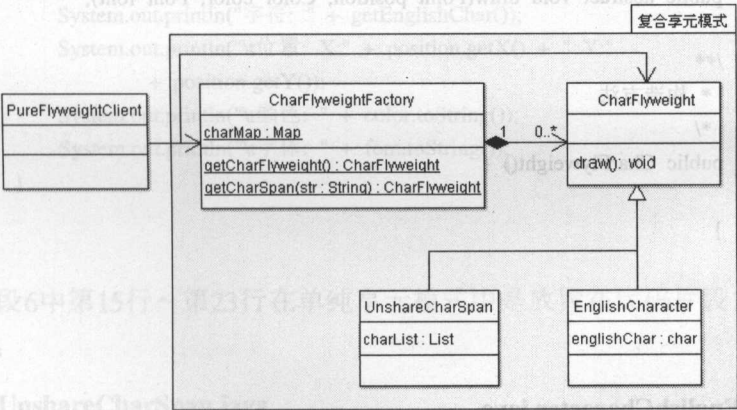


图15-5 复合享元模式UML类图

注意图15-3和图15-5有几处不同，CharFlyweightFactory中增加了一个创建UnshareCharSpan对象的方法，CharFlyweight中移除了内蕴状态的属性，将其转移到了具体的EnglishCharacter类中。其中的参与者的角色如下：

- CharFlyweight: 此抽象类的作用是提供享元类的接口。
- EnglishCharacter: 具体实现的享元类，提供了内蕴状态及外蕴状态的具体操作方法，通常并不会存储具体的外蕴状态数据。
- UnshareCharSpan: 不共享的享元类，其内部通常会维持一个享元对象的集合，一般采用组合模式设计其结构。
- CharFlyweightFactory: 享元管理对象，此类是保证享元对象被共享的关键类，通常使用对象池的方式存储享元对象。此类还提供了复合享元类的创建方法。
- PureFlyweightClient: 操作享元对象的客户类，注意此类使用享元对象时必须通过享元工厂类，它可以计算或存储享元对象的外蕴状态。

下面来具体看一下实现代码。首先是享元抽象类：

代码片段5 CharFlyweight.java

```

1 package cn.steven.pattern.demo.flyweight.composite;
2
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.Point;
6
7 /**
8  * 单纯享元模式抽象类
9  */
10 public abstract class CharFlyweight {
11     /**
12      * 外蕴状态
13      * @param position 字符位置
14      * @param color 字符颜色
15      * @param size 字号大小
16      * @param font 字体
17      */
18     public abstract void draw(Point position, Color color, Font font);
19
20     /**
21      * 构造方法
22      */
23     public CharFlyweight() {
24
25     }
26
27 }

```

单纯享元类：

代码片段6 EnglishCharacter.java

```

1 package cn.steven.pattern.demo.flyweight.composite;
2
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.Point;
6
7 /**
8  * 具体字符类
9  */
10 public class EnglishCharacter extends CharFlyweight {
11
12     /**
13      * 内蕴状态
14      */
15     private char englishChar;
16
17     public char getEnglishChar() {

```

```

18         return englishChar;
19     }
20
21     public void setEnglishChar(char englishChar) {
22         this.englishChar = englishChar;
23     }
24
25     /**
26      * 设置内蕴状态
27      *
28      * @param c
29      */
30     public EnglishCharacter(char c) {
31         this.setEnglishChar(c);
32     }
33
34     /**
35      * 设置外蕴状态
36      */
37     @Override
38     public void draw(Point position, Color color, Font font) {
39         System.out.println("字符: " + getEnglishChar());
40         System.out.println("t位置: X:" + position.getX() + " Y:"
41             + position.getY());
42         System.out.println("t颜色: " + color.toString());
43         System.out.println("t字体: " + font.toString());
44     }
45
46 }

```

注意代码片段6中第15行~第23行在单纯享元模式中是放置在代码片段1中的。

复合享元类:

代码片段7 UnshareCharSpan.java

```

1 package cn.steven.pattern.demo.flyweight.composite;
2
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.Point;
6 import java.util.List;
7
8 /**
9  * 复合享元模式类
10  * 注意此类不强制共享，通常是非共享的
11  * 它的作用是描述一段文字，这段文字有相同的外蕴属性
12  */
13 public class UnshareCharSpan extends CharFlyweight {
14
15     /**
16      * 内蕴状态
17      */

```

```

18     private List<CharFlyweight> charList;
19
20     public List<CharFlyweight> getCharList() {
21         return charList;
22     }
23
24     public void setCharList(List<CharFlyweight> charList) {
25         this.charList = charList;
26     }
27
28     /**
29      * 设置外蕴状态
30      */
31     @Override
32     public void draw(Point position, Color color, Font font) {
33         /**
34          * 递归使内部的享元都设置外蕴状态
35          */
36         for (CharFlyweight charFlyweight : charList) {
37             charFlyweight.draw(position, color, font);
38         }
39     }
40
41 }

```

注意代码片段7中的内蕴状态是一个享元集合，当对其设置外蕴状态时，它会将集合中的对象都进行外蕴状态的设置。

享元工厂类：

代码片段8 CharFlyweightFactory.java

```

1  package cn.steven.pattern.demo.flyweight.composite;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.List;
6  import java.util.Map;
7
8  /**
9   * 享元模式的工厂方法
10  */
11  public class CharFlyweightFactory {
12
13      /**
14       * Flyweight对象缓存池
15       */
16      private static Map<Character, CharFlyweight> charMap =
17          new HashMap<Character, CharFlyweight>();
18
19      public synchronized static CharFlyweight getCharFlyweight(char c) {
20          /**

```



```

21      * 查看此对象是否已经存在
22      */
23      if (charMap.containsKey(c)) {
24          // 存在此对象则直接使用原有对象
25          return charMap.get(c);
26      } else {
27          // 不存在此对象生成新对象
28          CharFlyweight charFlyweight = new EnglishCharacter(c);
29          // 存入集合中
30          charMap.put(c, charFlyweight);
31          return charFlyweight;
32      }
33  }
34
35  public synchronized static CharFlyweight getCharSpan(String str) {
36      UnshareCharSpan charSpan = new UnshareCharSpan();
37      List<CharFlyweight> charList = new ArrayList<CharFlyweight>();
38      /**
39       * 对str中的每一个子符都只用共享
40       */
41      for (int i = 0; i < str.length(); i++) {
42          charList.add(getCharFlyweight(str.charAt(i)));
43      }
44      /**
45       * 保存内蕴状态
46       */
47      charSpan.setCharList(charList);
48      return charSpan;
49  }
50
51  /**
52   * 显示Flyweight对象缓存池状态
53   */
54  public static void showStatus() {
55      System.out.print("flyweight对象数量: " + charMap.size()
56          + " 字符列表: ");
57      for (Map.Entry<Character, CharFlyweight> entry : charMap
58          .entrySet()) {
59          System.out.print(entry.getKey() + " ");
60      }
61      System.out.println();
62  }
63  }

```

注意代码片段8中第35行~第49行代码生成了一个UnshareCharSpan对象, 这个对象中使用的是共享的享元字符对象。

客户端代码:

代码片段9 PureFlyweightClient.java

```

1  package cn.steven.pattern.demo.flyweight.composite;
2

```

```
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.Point;
6
7 /**
8  * 复合享元模式测试类
9  */
10 public class PureFlyweightClient {
11     /**
12      * 显示Java字符串
13      */
14     public static void main(String[] args) {
15
16         // 显示Flyweight对象缓存池状态
17         CharFlyweightFactory.showStatus();
18         // 创建复合享元对象
19         CharFlyweight str = CharFlyweightFactory.getCharSpan("java");
20         // 设置外蕴状态
21         str.draw(new Point(2, 5), Color.red, new Font(
22             Font.MONOSPACED, Font.PLAIN, 8));
23         CharFlyweightFactory.showStatus();
24     }
25
26 }
```

运行结果如图15-6所示。



图15-6 PureFlyweightClient.java运行结果

结合代码片段9和图15-6可以看出，在应用中创建的UnshareCharSpan对象中使用的字符的确是共享的，但是UnshareCharSpan对象本身不是共享的，这就是复合享元模式的典型应用场景。

15.2.3 使用享元模式解决宣传海报的设计问题

经过了外观模式的学习，现在可以思考一下宣传海报的设计问题了，由图15-1和图15-2可见，构成海报的主要元素是字符和图片，而这两者都应该是可以共享的，同时我们还需要设计一个可以容纳享元对象并维护其外蕴状态的类，也就是非共享的类来进行对象管理。在设计过

程中，状态的运算需要外部化，所以还需要设计一个上下文状态类，参考设计图如图15-7所示。

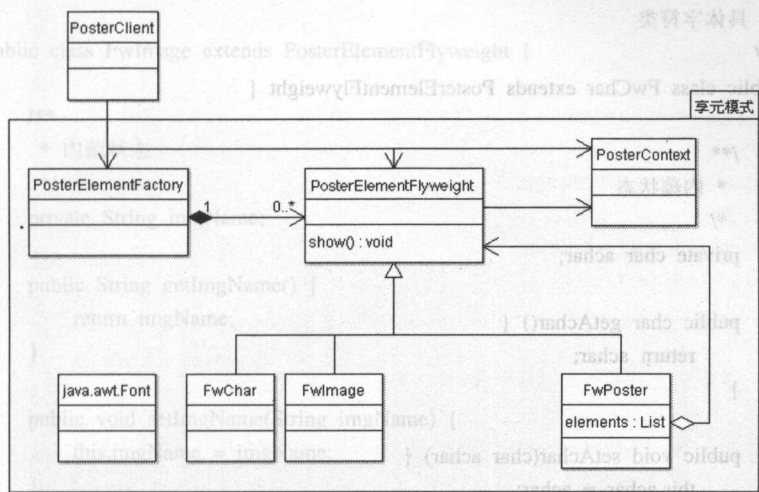


图15-7 海报问题设计图

注意，图15-7中FwPoster是一个复合享元类，通常不可共享，它维护了其子元素的外蕴状态。首先看一下享元抽象类的代码：

代码片段10 PosterElementFlyweight.java

```
1 package cn.steven.pattern.demo.flyweight.example;
2
3 import java.awt.Dimension;
4 import java.awt.Font;
5
6 /**
7  * 海报元素抽象类
8  */
9 public abstract class PosterElementFlyweight {
10
11     /**
12      * 显示元素
13      * @param font 需要使用的字体， 图片元素可设置为null
14      * @param dimension 显示尺寸， 字符元素可设置为null
15      * @param ctx 显示的上下文环境
16      */
17     abstract public void draw(Font font, Dimension dimension,
18                               PosterContext ctx);
19 }
```

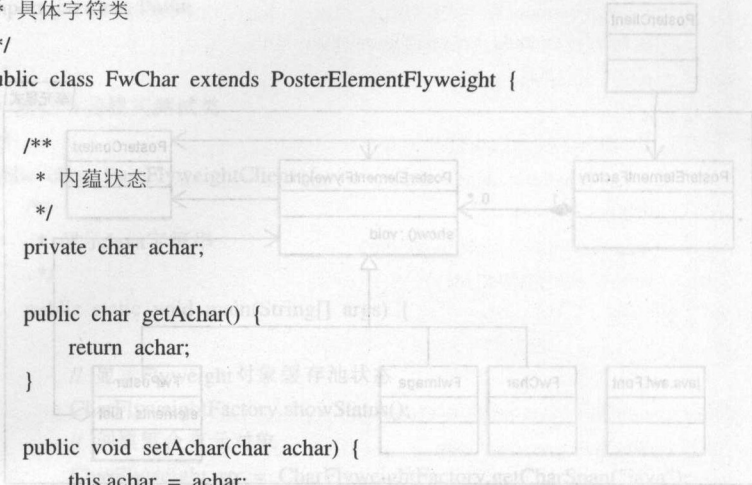
字符享元类：

代码片段11 FwChar.java

```
1 package cn.steven.pattern.demo.flyweight.example;
2
3 import java.awt.Dimension;
4 import java.awt.Font;
5 import java.awt.FontMetrics;
```



```
6
7 /**
8  * 具体字符类
9  */
10 public class FwChar extends PosterElementFlyweight {
11
12     /**
13      * 内蕴状态
14      */
15     private char achar;
16
17     public char getAchar() {
18         return achar;
19     }
20
21     public void setAchar(char achar) {
22         this.achar = achar;
23     }
24
25     /**
26      * 设置内蕴状态
27      *
28      * @param c
29      */
30     public FwChar(char c) {
31         this.setAchar(c);
32     }
33
34     /**
35      * 设置外蕴状态
36      */
37     @Override
38     public void draw(Font font, Dimension dimension, PosterContext ctx) {
39         // 输出字符
40         System.out.println("字符: " + getAchar());
41         System.out.println("\t位置: X:" + ctx.getNowPosition().x
42             + " Y:" + ctx.getNowPosition().y);
43         System.out.println("\t字体: " + font.toString());
44         // 设置上下文状态
45         FontMetrics fm = sun.font.FontDesignMetrics.getMetrics(font);
46         ctx.put(new Dimension(fm.charWidth(getAchar()), fm
47             .getHeight()));
48     }
49
50 }
```



图片享元类:

代码片段12 FwImage.java

```
1 package cn.steven.pattern.demo.flyweight.example;
2
```

```

3 import java.awt.Dimension;
4 import java.awt.Font;
5
6 public class FwImage extends PosterElementFlyweight {
7
8     /**
9      * 内蕴状态
10     */
11     private String imgName;
12
13     public String getImgName() {
14         return imgName;
15     }
16
17     public void setImgName(String imgName) {
18         this.imgName = imgName;
19     }
20
21     /**
22      * 构造方法
23      *
24      * @param imgName
25      *      图片名称
26      */
27     public FwImage(String imgName) {
28         this.imgName = imgName;
29     }
30
31     /**
32      * 设置外蕴状态
33      */
34     @Override
35     public void draw(Font font, Dimension dimension, PosterContext ctx) {
36         // 显示图片
37         System.out.println("图片:" + getImgName());
38         System.out.println("\t位置: X:" + ctx.getNowPosition().x
39             + " Y:" + ctx.getNowPosition().y);
40         System.out.println("\t大小: w:" + dimension.width + " h:"
41             + dimension.height);
42         // 设置上下文状态
43         ctx.put(dimension);
44     }
45
46 }

```

注意代码片段11和代码片段12这两种类分别对应两种不同的享元对象，它们有各自的内蕴数据和外蕴数据，比如FwChar的内蕴数据是字符，外蕴数据中最重要的是字体，而FwImage的内蕴数据是文件名（在这里是为了简化数据模型，其实应该是图形对象），外蕴数据中最重要的是图形尺寸。

为了使这两种完全不同的数据结构能在一个抽象层次中工作，在代码片段10中设计的draw

方法同时包含了它们各自需要的外蕴数据。

海报复合享元类：

代码片段13 FwPoster.java

```
1 package cn.steven.pattern.demo.flyweight.example;
2
3 import java.awt.Dimension;
4 import java.awt.Font;
5 import java.util.LinkedList;
6 import java.util.List;
7
8 /**
9  * 复合享元类，通常不用于共享
10 */
11 public class FwPoster extends PosterElementFlyweight {
12     /**
13      * 元素列表
14      */
15     private List<PosterElementFlyweight> elelist;
16
17     /**
18      * 元素使用的字体
19      */
20     private Font font;
21
22     /**
23      * 保存上下文
24      */
25     private PosterContext ctx;
26
27     public PosterContext getCtx() {
28         return ctx;
29     }
30
31     public void setCtx(PosterContext ctx) {
32         this.ctx = ctx;
33     }
34
35     /**
36      * 构造方法
37      */
38     public FwPoster() {
39         this.elelist = new LinkedList<PosterElementFlyweight>();
40     }
41
42     public List<PosterElementFlyweight> getElelist() {
43         return elelist;
44     }
45
46     public void setElelist(List<PosterElementFlyweight> elelist) {
```



```

47         this.elelist = elelist;
48     }
49
50     public Font getFont() {
51         return font;
52     }
53
54     public void setFont(Font font) {
55         this.font = font;
56     }
57
58     public Dimension getDimension() {
59         return dimension;
60     }
61
62     public void setDimension(Dimension dimension) {
63         this.dimension = dimension;
64     }
65
66     /**
67      * 元素大小，供图片使用
68      */
69     private Dimension dimension;
70
71     /**
72      * 绘画方法
73      */
74     @Override
75     public void draw(Font font, Dimension dimension, PosterContext ctx) {
76         // 如果自身已有内蕴属性，则用自己的，否则传入的
77         this.font = (this.font == null ? font : this.font);
78         this.dimension = (this.dimension == null ? dimension
79             : this.dimension);
80         this.ctx = (this.ctx == null ? ctx : this.ctx);
81         // 操作集合中的对象
82         for (PosterElementFlyweight e : elelist) {
83             e.draw(this.font, this.dimension, this.ctx);
84         }
85     }
86
87     /**
88      * 设置内蕴状态
89      * @param font 字体
90      * @param dimension 尺寸
91      * @param ctx 上下文环境
92      */
93     public void setStatus(Font font, Dimension dimension,
94         PosterContext ctx) {
95         this.font = font;
96         this.dimension = dimension;
97         this.ctx = ctx;

```

```

98     }
99
100 }

```

代码片段13所示的类是享元结构中最复杂的类，此类的作用是采用了组合的方式管理一个树状的结构，每一个节点都是一个PosterElementFlyweight。

享元工厂：

代码片段14 PosterElementFactory.java

```

1  package cn.steven.pattern.demo.flyweight.example;
2
3  import java.awt.Font;
4  import java.util.HashMap;
5  import java.util.HashSet;
6  import java.util.Map;
7  import java.util.Set;
8
9  /**
10   * 享元模式的工厂方法
11   */
12  public class PosterElementFactory {
13
14      /**
15       * 单例模式
16       */
17      private static PosterElementFactory fac;
18
19      /**
20       * 单例模式实例获取方法
21       */
22      public synchronized static PosterElementFactory getFactory() {
23          if (fac == null) {
24              fac = new PosterElementFactory();
25          }
26          return fac;
27      }
28
29      /**
30       * 单例模式构造方法
31       */
32      private PosterElementFactory() {
33      }
34
35      /**
36       * 享元元素的类型
37       */
38      public static final String ElementType_Charactor = "Charactor";
39      public static final String ElementType_Image = "Image";
40
41      /**
42       * Flyweight对象缓存池

```

```

43     */
44     private Map<String, Map<Object, PosterElementFlyweight>> fwMap
45         = new HashMap<String, Map<Object, PosterElementFlyweight>>();
46
47     /**
48      * font对象缓存池
49      */
50     private Set<Font> fontSet = new HashSet<Font>();
51
52     /**
53      * 获取字体对象
54      */
55     public synchronized Font getFont(Font font){
56         /**
57          * 查看此对象是否已经存在
58          */
59         if(fontSet.contains(font)){
60             // 存在此对象则直接使用原有对象
61             for (Font f : fontSet) {
62                 if(f.equals(font)){
63                     return f;
64                 }
65             }
66         }else{
67             // 不存在此对象则存入此对象
68             fontSet.add(font);
69         }
70         return font;
71     }
72
73     /**
74      * 获取一个享元对象
75      *
76      * @param ElementType 对象类型
77      * @param key 享元对象的创建值
78      */
79     public synchronized PosterElementFlyweight getFw(
80         String ElementType, Object key) {
81         /**
82          * 查看此对象是否已经存在
83          */
84         if (fwMap.get(ElementType).containsKey(key)) {
85             // 存在此对象则直接使用原有对象
86             return fwMap.get(ElementType).get(key);
87         } else {
88             // 不存在此对象则生成新对象
89             PosterElementFlyweight fwObj = null;
90             if (PosterElementFactory.ElementType_Charactor
91                 .equals(ElementType)) {
92                 fwObj = new FwChar((Character) key);
93             } else if (PosterElementFactory.ElementType_Image

```



```

94         .equals(ElementType)) {
95             fwObj = new FwImage((String) key);
96         } else {
97             throw new RuntimeException("不支持此类对象!");
98         }
99         // 存入集合中
100         fwMap.get(ElementType).put(key, fwObj);
101         return fwObj;
102     }
103 }
104 }
105
106 /**
107  * 创建非共享享元对象
108  *
109  * @param ElementType 对象类型
110  * @param font 字体
111  * @param dimension 尺寸
112  * @param str 对象信息
113  */
114 public synchronized FwPoster getPoster(String ElementType,
115     String str) {
116
117     if(fwMap.get(ElementType) == null){
118         fwMap.put(ElementType,
119             new HashMap<Object, PosterElementFlyweight>());
120     }
121
122     FwPoster fwPoster = new FwPoster();
123     if (PosterElementFactory.ElementType_Character
124         .equals(ElementType)) {
125         for (int i = 0; i < str.length(); i++) {
126             fwPoster.getElelist().add(getFw(
127                 PosterElementFactory.ElementType_Character,
128                 str.charAt(i)));
129         }
130     }
131     } else if (PosterElementFactory.ElementType_Image
132         .equals(ElementType)) {
133         fwPoster.getElelist().add(getFw(
134             PosterElementFactory.ElementType_Image, str));
135     } else {
136         throw new RuntimeException("不支持此类对象!");
137     }
138     return fwPoster;
139 }
140 }
141
142 /**
143  * 显示Flyweight对象缓存池状态
144  */

```

```
145 public void showStatus() {
146     if(fwMap.get(ElementType_Charactor)!=null){
147         System.out.print("字符对象数量: " +
148             fwMap.get(ElementType_Charactor).size());
149     }
150     if(fwMap.get(ElementType_Image)!=null){
151         System.out.print("图片对象数量: " +
152             fwMap.get(ElementType_Image).size());
153     }
154     System.out.println(fontSet.size());
155 }
156 }
```

注意代码片段14中的复杂之处:

- 代码14行~33行: 单例模式的实现, 最常见的设计就是将工厂类设计为单例的。
- 代码41行~45行: 保存享元对象的集合, 在这里使用了Map的嵌套泛型。
- 代码47行~71行: 字体对象的共享实现。
- 代码79行~104行: 获取享元对象方法, 此处需要判断是哪一种享元对象。
- 代码106行~140行: 创建非共享享元对象。

上下文对象:

代码片段15 PosterContext.java

```
1 package cn.steven.pattern.demo.flyweight.example;
2
3 import java.awt.Dimension;
4 import java.awt.Point;
5
6 /**
7  * 海报上下文
8  */
9 public class PosterContext {
10     /**
11      * 纸张大小
12      */
13     private Dimension pageSize;
14
15     /**
16      * 当前位置
17      */
18     private Point nowPosition;
19
20     /**
21      * 构造方法
22      *
23      * @param pageSize 纸张大小
24      */
25     public PosterContext(Dimension pageSize) {
26         this.pageSize = pageSize;
27         // 设置默认位置
```

```

28         this.setNowPosition(new Point(0, 0));
29     }
30
31     /**
32      * 放置一个元素, 当前位置应移动
33      *
34      * @param dimension 元素尺寸
35      */
36     public void put(Dimension dimension) {
37         // 移动x坐标
38         this.getNowPosition().x = this.getNowPosition().x
39             + dimension.width;
40         if (this.getNowPosition().x < 0
41             || this.getNowPosition().x > pageSize.width) {
42             throw new RuntimeException("超出纸张边界");
43         }
44     }
45
46     public Dimension getPageSize() {
47         return pageSize;
48     }
49
50     public void setPageSize(Dimension pageSize) {
51         this.pageSize = pageSize;
52     }
53
54     public Point getNowPosition() {
55         return nowPosition;
56     }
57
58     public void setNowPosition(Point nowPosition) {
59         this.nowPosition = nowPosition;
60     }
61
62 }

```

此类的作用是保存和控制当前的显示环境。

客户端代码:

代码片段16 PosterClient.java

```

1 package cn.steven.pattern.demo.flyweight.example;
2
3 import java.awt.Dimension;
4 import java.awt.Font;
5
6 public class PosterClient {
7
8     /**
9      * 测试享元对象
10      */
11     public static void main(String[] args) {

```



```
12
13 // 生成上下文, 设置纸张大小
14 PosterContext ctx = new PosterContext(
15     new Dimension(1024, 768));
16
17 // 创建工厂
18 PosterElementFactory fac = PosterElementFactory.getFactory();
19
20 // 创建一个文本区域
21 FwPoster textSpan_1 = fac.getPoster(
22     PosterElementFactory.ElementType_Character, "中文");
23
24 textSpan_1.setStatus(fac.getFont(new Font("宋体", Font.PLAIN,
25     12)), new Dimension(0, 0), ctx);
26
27 // 创建图片区域
28 FwPoster imageSpan_1 = fac.getPoster(
29     PosterElementFactory.ElementType_Image, "image_1");
30 imageSpan_1.setStatus(null, new Dimension(100, 200), ctx);
31
32 // 创建一个文本区域
33 FwPoster textSpan_2 = fac.getPoster(
34     PosterElementFactory.ElementType_Character, "zoo");
35
36 textSpan_2.setStatus(fac.getFont(new Font("Century",
37     Font.BOLD, 12)), new Dimension(0, 0), ctx);
38
39 // 创建图片区域
40 FwPoster imageSpan_2 = fac.getPoster(
41     PosterElementFactory.ElementType_Image, "image_1");
42 imageSpan_2.setStatus(null, new Dimension(50, 100), ctx);
43
44 // 创建组合对象
45 FwPoster compPoster = new FwPoster();
46 compPoster.getElelist().add(textSpan_1);
47 compPoster.getElelist().add(imageSpan_1);
48 compPoster.getElelist().add(textSpan_2);
49 compPoster.getElelist().add(imageSpan_2);
50
51 // 打印, 可以不给出各种参数, 意为使用组合的内蕴状态
52 compPoster.draw(null, null, null);
53 }
54
55 }
```

注意代码片段16中创建的对象结构如图15-8所示。

由图15-8可以看出, FwPoster对象可以组合任意一种PosterElementFlyweight的子类对象, 这对于制作复杂的海报是必不可少的功能, 图中还可以看出, 不同的FwPoster对象中可以放置相同的对象, 以达到共享的目的。

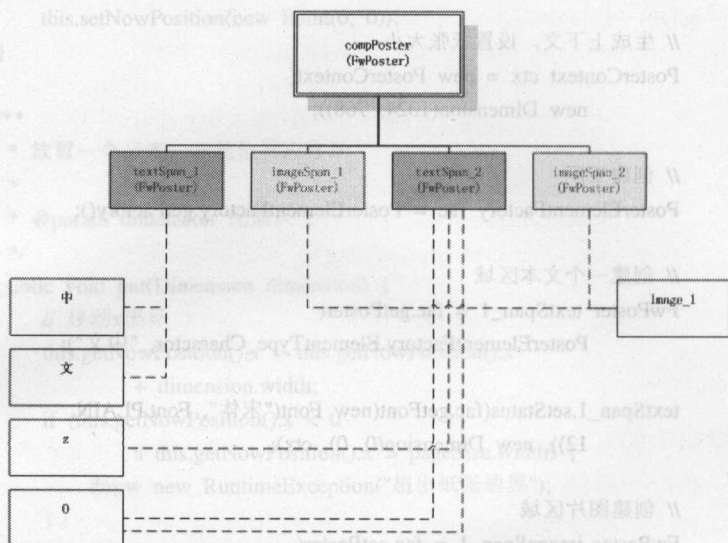


图15-8 PosterClient.java对象结构

其运行结果如图15-9所示。

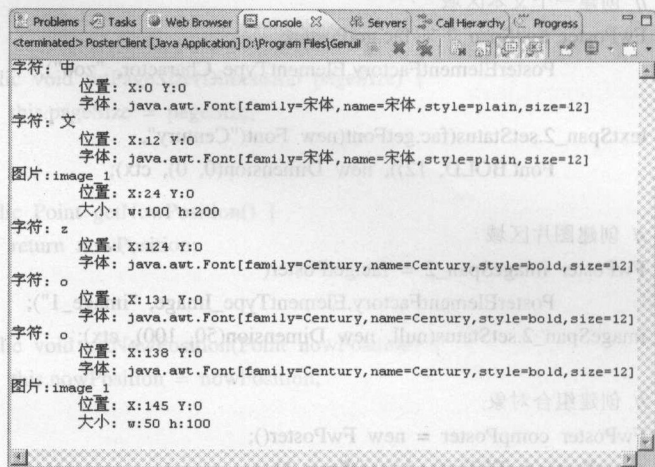


图15-9 PosterClient.java运行结果

由图15-9可以看出：

- 字体是共享的。
- 字符对象可以共享。
- 图片对象可以共享。
- 外部状态可以自行通过计算改变（位置x坐标）。

在设计享元模式时，一定要进行对象的共享，这样可以减少对象数目以及内存的开销。

15.2.4 享元模式在JDK中的实例

享元模式在Java的设计中比较常见，但是通常客户应用代码中却不知道后台使用的是此模式，因为通常享元模式的接口应用比较复杂，需要简化。

首先来看一下一个关于String类的例子：

代码片段17 TestString.java

```
1 package cn.steven.pattern.demo.flyweight.jdk;
2
3 /**
4  * 测试String对象的享元模式
5  */
6 public class TestString {
7     public static void main(String[] args) {
8         String a = "abc";
9         String b = "abc";
10        // 下面会输出true
11        System.out.println(a == b);
12    }
13 }
```

注意上例中对象a和b指向的是内存中的同一个地址！也就是说a和b共享了同一个字符串对象。如果想理解这个问题，首先要知道Java是如何处理String对象的。

Java内部将创建String对象的语句转化为以下几个步骤：

- (1) 先定义一个名为a的对String类的对象引用变量：String a;
- (2) 在栈中查找有没有存放值为“abc”的地址，如果没有，则开辟一个存放值为“abc”的地址，接着创建一个新的String类的对象obj，并将obj的字符串值指向这个地址，而且在栈中这个地址旁边记下这个引用的对象obj。如果已经有了值为“abc”的地址，则查找对象obj，并返回obj的地址。
- (3) 将a指向对象o的地址。

注意第2步是典型的享元模式思想，所以String对象才能出现共享的效果。注意，在Java和C#中的字符串对象都被设计为了不可变对象。这样做更加方便了享元模式的实现，但是在字符串修改时就会有性能上的影响了，解决办法是如果使用的字符串需要频繁改动，则需要使用其非享元的类StringBuffer。

为了比较String和StringBuffer类在字符串修改上的效率，请看以下测试类：

代码片段18 TestString2.java

```
1 package cn.steven.pattern.demo.flyweight.jdk;
2 /**
3  * 测试String与StringBuffer的效率
4  */
5 public class TestString2 {
6
7     public static void main(String[] args) {
8         long time = System.currentTimeMillis();
9         String a = "";
10        for (int i = 0; i < 10000; i++) {
11            a += i;
12        }
13        System.out.println("String 用时 : "
14            + (System.currentTimeMillis() - time) + "ms");
15        time = System.currentTimeMillis();
16        StringBuffer sb = new StringBuffer();
```



```
17         for (int i = 0; i < 10000; i++) {
18             sb.append(i);
19         }
20         System.out.println("StringBuffer 用时 : "
21             + (System.currentTimeMillis() - time) + "ms");
22     }
23
24 }
```

运行结果如图15-10所示。

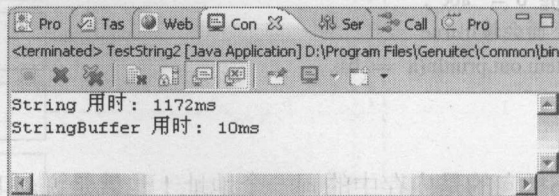


图15-10 TestString2.java运行结果

注意其应用在10000次修改时效率已经相差100倍以上！

所以，使用享元模式一定要根据需求而定，不变的对象才适合共享，变动的则不适合。

15.2.5 享元模式的使用范围

Flyweight模式的有效性很大程度上取决于在何种情景下使用它。当以下情况都成立时可使用Flyweight模式：

- 一个应用程序使用了大量的对象。
- 完全由于使用大量的对象，造成了很大的存储开销。
- 对象的大多数状态都可变为外部状态。
- 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
- 应用程序不依赖对象的唯一性标识。

享元模式的主要缺点如下：

- 享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使程序的逻辑复杂化。
- 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

15.2.6 与其他模式的关系

创建享元对象通常需要用工厂模式，此工厂通常被设计为单例模式。

享元模式有对象唯一性，但是它与单例模式不同，使用单例模式不能在外部实例化，但是享元模式可以。单例模式给外部提供了一致的服务，享元模式由于外部可以提供外蕴状态，所以可以提供不同的服务。

享元工厂中保存了享元对象与它们所属的对象的映射表，这其中使用了备忘录模式。

复合享元模式中使用了组合模式。

15.3 享元模式总结

享元模式 (Flyweight) 运用共享技术有效地支持了大量细粒度的对象。它适合在程序开发中出现大量相似对象时使用。使用此模式时要注意其应用的场景, 虽然使用它可以有效地减少资源的开销, 但是如果使用不当将适得其反。

实现享元模式时其缓存的查询效率是影响性能的关键点。

处理咨询的

代码片段2

ServiceManager.java

图 15-1 系统架构图

```
1 package cn.steven.pattern.demo.command.request;
```

此图展示了系统架构。图中包含一个名为“ServiceManager”的类，它负责管理“ServiceType”和“Service”对象。图中还显示了“ServiceType”和“Service”对象的实例化过程。图中还显示了“ServiceType”和“Service”对象的实例化过程。

```
5 //
```

图中还显示了“ServiceType”和“Service”对象的实例化过程。图中还显示了“ServiceType”和“Service”对象的实例化过程。

```
10
```

```
System.out.println("ServiceManager");
```

图中还显示了“ServiceType”和“Service”对象的实例化过程。图中还显示了“ServiceType”和“Service”对象的实例化过程。

```
12
```

服务的枚举:

图 15.1 系统架构图

代码片段3 ServiceType.java

图中还显示了“ServiceType”和“Service”对象的实例化过程。图中还显示了“ServiceType”和“Service”对象的实例化过程。

```
3 /**
```

```
4 * 服务类型
```

```
5 */
```

```
6 public enum ServiceType {
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15
```

```
16
```

```
17
```

```
18
```

```
19
```

```
20
```

```
21
```

```
22
```

```
23
```

```
24
```

```
25
```

```
26
```

```
27
```

```
28
```

```
29
```

```
30
```

```
31
```

```
32
```

```
33
```

```
34
```

```
35
```

```
36
```

```
37
```

```
38
```

```
39
```

```
40
```

```
41
```

```
42
```

```
43
```

```
44
```

```
45
```

```
46
```

```
47
```

```
48
```

```
49
```

```
50
```

```
51
```

```
52
```

```
53
```

```
54
```

```
55
```

```
56
```

```
57
```

```
58
```

```
59
```

```
60
```

```
61
```

```
62
```

```
63
```

```
64
```

```
65
```

```
66
```

```
67
```

```
68
```

```
69
```

```
70
```

```
71
```

```
72
```

```
73
```

```
74
```

```
75
```

```
76
```

```
77
```

```
78
```

```
79
```

```
80
```

```
81
```

```
82
```

```
83
```

```
84
```

```
85
```

```
86
```

```
87
```

```
88
```

```
89
```

```
90
```

```
91
```

```
92
```

```
93
```

```
94
```

```
95
```

```
96
```

```
97
```

```
98
```

```
99
```

```
100
```

```
101
```

```
102
```

```
103
```

```
104
```

```
105
```

```
106
```

```
107
```

```
108
```

```
109
```

```
110
```

```
111
```

```
112
```

```
113
```

```
114
```

```
115
```

```
116
```

```
117
```

```
118
```

```
119
```

```
120
```

```
121
```

```
122
```

```
123
```

```
124
```

```
125
```

```
126
```

```
127
```

```
128
```

```
129
```

```
130
```

```
131
```

```
132
```

```
133
```

```
134
```

```
135
```

```
136
```

```
137
```

```
138
```

```
139
```

```
140
```

```
141
```

```
142
```

```
143
```

```
144
```

```
145
```

```
146
```

```
147
```

```
148
```

```
149
```

```
150
```

```
151
```

```
152
```

```
153
```

```
154
```

```
155
```

```
156
```

```
157
```

```
158
```

```
159
```

```
160
```

```
161
```

```
162
```

```
163
```

```
164
```

```
165
```

```
166
```

```
167
```

```
168
```

```
169
```

```
170
```

```
171
```

```
172
```

```
173
```

```
174
```

```
175
```

```
176
```

```
177
```

```
178
```

```
179
```

```
180
```

```
181
```

```
182
```

```
183
```

```
184
```

```
185
```

```
186
```

```
187
```

```
188
```

```
189
```

```
190
```

```
191
```

```
192
```

```
193
```

```
194
```

```
195
```

```
196
```

```
197
```

```
198
```

```
199
```

```
200
```

```
201
```

```
202
```

```
203
```

```
204
```

```
205
```

```
206
```

```
207
```

```
208
```

```
209
```

```
210
```

```
211
```

```
212
```

```
213
```

```
214
```

```
215
```

```
216
```

```
217
```

```
218
```

```
219
```

```
220
```

```
221
```

```
222
```

```
223
```

```
224
```

```
225
```

```
226
```

```
227
```

```
228
```

```
229
```

```
230
```

```
231
```

```
232
```

```
233
```

```
234
```

```
235
```

```
236
```

```
237
```

```
238
```

```
239
```

```
240
```

```
241
```

```
242
```

```
243
```

```
244
```

```
245
```

```
246
```

```
247
```

```
248
```

```
249
```

```
250
```

```
251
```

```
252
```

```
253
```

```
254
```

```
255
```

```
256
```

```
257
```

```
258
```

```
259
```

```
260
```

```
261
```

```
262
```

```
263
```

```
264
```

```
265
```

```
266
```

```
267
```

```
268
```

```
269
```

```
270
```

```
271
```

```
272
```

```
273
```

```
274
```

第16章 命令模式 (Command)

命令模式的意图是将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化，对请求进行排队或记录下请求的日志，以及支持可以撤销的操作¹。

命令模式在设计模式分组中属于一种行为模式，命令模式的核心在于把模块之间的多种调用进行规范和抽象，就如公司中上级对下级发布命令的过程，比如有一个领导在国外，他需要向不同部门发出不同的工作要求（称为命令），再假设他没有方便的通信工具，只能通过信件来传达他的要求，那么他将会把给不同部门的要求写成一封信，这样所有的命令都具有了信件的共同特征，而邮递员并不了解信件的内容是什么，他的任务只是负责传递命令，而每个部门的负责人都知道如何打开这个信件和执行信中所表达的任务要求。把这样一个类似问题进行抽象，就成了这里所提到的命令模式。

上面的一个领导下达命令的例子是一种封装命令的典型示例，计算机软件中也有很多这样的案例，比如一个媒体播放器如mplayer²，这种开源的播放器软件可以支持多种视频格式，如rmvb、mov、wmv、mp4、flv等，这些格式的编码方式很显然是不一样的。mplayer播放控制条如图16-1所示。



图16-1 播放控制条

客户只需要根据需要操作这些控制按钮即可，比如单击播放按钮之后，mplayer会正确执行将这个命令，实际上后台它将调用特定的解码功能，但是这个过程对于客户是透明的，这样就极大地方便了客户的使用。

命令模式还有一个特点就是支持撤销和恢复，这个功能在代码编辑时是非常常用的，比如输入了一段代码，发现其实没有必要，那么单击“撤销”功能按钮即可，过了一会仔细想了想发现写上还是有意义的，于是单击“恢复”按钮，消失的代码又回来了。功能全面的编辑器还包括无限“撤销”功能，以及宏命令（就是一组命令的集合）功能。

以上的例子中全部使用了命令模式，由此可见，命令模式是常用的一个关键模式。

16.1 连锁店客服专线的问题

随着连锁店的生意越做越大，分店也越来越多，为了解决顾客的各种问题，连锁店总店开通了一部客服专线“88812345678”，这样，顾客的各种需求，比如投诉、问询、团购等，就可以通过拨打这个电话来解决了。

在拨打电话的过程中有几种处理问题的方法：

¹Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.GOF[95]

²<http://www.mplayerhq.hu/>.

- 接线员听电话并处理问题。
- 接线员找到可以处理问题的人并把电话交由此人处理问题。
- 接线员找到分配任务的人接线, 由他来分配到具体的人处理问题。
- 接线员保留顾客的电话录音并交给处理的人解决问题。

下面来看一下简单的编程方法产生的问题。

处理投诉的人员:

代码片段1 ServiceManA.java

```
1 package cn.steven.pattern.demo.command.quest;
2
3 /**
4  * 处理投诉的人
5  */
6 public class ServiceManA {
7
8     public void answer(){
9         System.out.println("了解问题");
10        System.out.println("处理投诉");
11    }
12 }
```

处理咨询的人:

代码片段2 ServiceManB.java

```
1 package cn.steven.pattern.demo.command.quest;
2
3 /**
4  * 咨询人
5  */
6 public class ServiceManB {
7
8     public void response(){
9         System.out.println("了解问题");
10        System.out.println("进行回答");
11    }
12 }
```

服务的枚举:

代码片段3 ServiceType.java

```
1 package cn.steven.pattern.demo.command.quest;
2
3 /**
4  * 服务类型
5  */
6 public enum ServiceType {
7
8     /**
9      * 类型的种类
10     */
11 }
```

```
10      */
11      投诉, 咨询
12  }
```

接线员:

代码片段4 PhoneMan.java

```
1 package cn.steven.pattern.demo.command.queue;
2
3 /**
4  * 接线员
5  */
6 public class PhoneMan {
7
8     public void service(ServiceType type) {
9         switch (type) {
10             case 投诉:
11                 new ServiceManA().answer();
12                 break;
13             case 咨询:
14                 new ServiceManB().response();
15                 break;
16             default:
17                 System.out.println("不支持此类操作");
18                 break;
19         }
20     }
21
22 }
```

客户代码:

代码片段5 Customer.java

```
1 package cn.steven.pattern.demo.command.queue;
2
3 /**
4  * 打客服电话的顾客
5  */
6 public class Customer {
7
8     public static void main(String[] args) {
9
10         /**
11          * 拨打电话至客服
12          */
13         PhoneMan pm = new PhoneMan();
14
15         /**
16          * 选择服务类型
17          */
18         pm.service(ServiceType.咨询);
```

```
19         pm.service(ServiceType.投诉);
20     }
21
22 }
```

上述解决方案的序列图如图16-2所示。

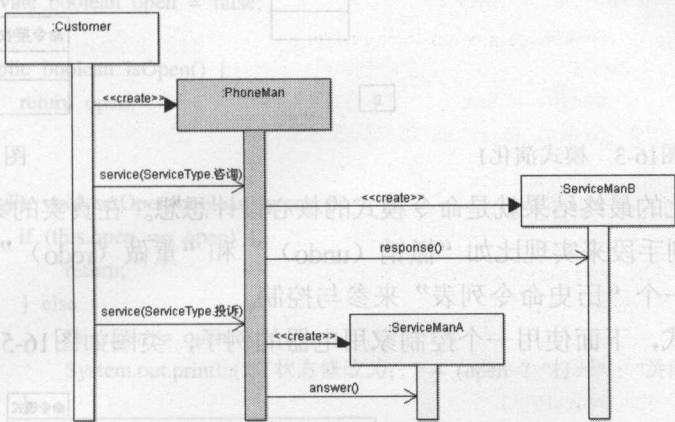


图16-2 客服专线的序列图

由以上代码和图16-2可见，程序可以正确运行，但是仔细分析后会发现以下问题：

- 新增客服种类困难。
- 接线员必须得知具体的问题处理人员进行业务处理的方法，新增处理人员时会更加不方便。
- 如果接线员没法立即联系到处理人员则服务停滞。
- 无法对业务进行撤销动作。

对于这种需要处理命令的情况，使用命令模式会很方便地解决上述问题。

16.2 命令模式的结构

命令模式对命令、具体命令、命令执行者等参与者都做了很好的分离设计，适用于复杂的命令处理系统的架构设计。

16.2.1 命令模式

仔细观察客服专线问题的初步实现就会发现一个问题，由于接线员要传递一种“命令”，但是这种命令并没有做到很好的封装，所以命令的实现者，命令的调用者都会出现使用起来较复杂的情况，如图16-3所示。

A图为问题解决时所采用的办法，这种办法的缺点前面已经讨论过了，根据面向对象的原则，可以将变化的地方进行封装，让命令调用者依赖命令接口而不是直接依赖实现者，这样就解决了A方法所出现的问题。

为了更好地在组件之间进行配合，B图中的命令实现者其实是一个简化设计，适应性更好的设计方法是将命令实现者和命令接收者再进行分离，如图16-4所示。

分离之后命令的实现就可以不直接和执行命令的代码耦合，从而增强了系统的灵活性。

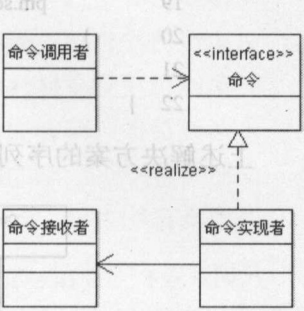
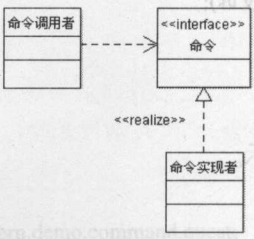
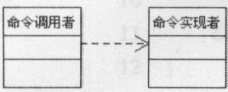


图16-3 模式演化1

图16-4 模式演化2

以上的模式演化的最终结果就是命令模式的核心设计思想。在真实的案例中，命令模式通常还需要用一些控制手段来实现比如“撤销（undo）”和“重做（redo）”的功能。命令的执行过程还需要使用一个“历史命令列表”来参与控制。

为了说明此模式，下面使用一个控制家用电器的例子，类图如图16-5所示。

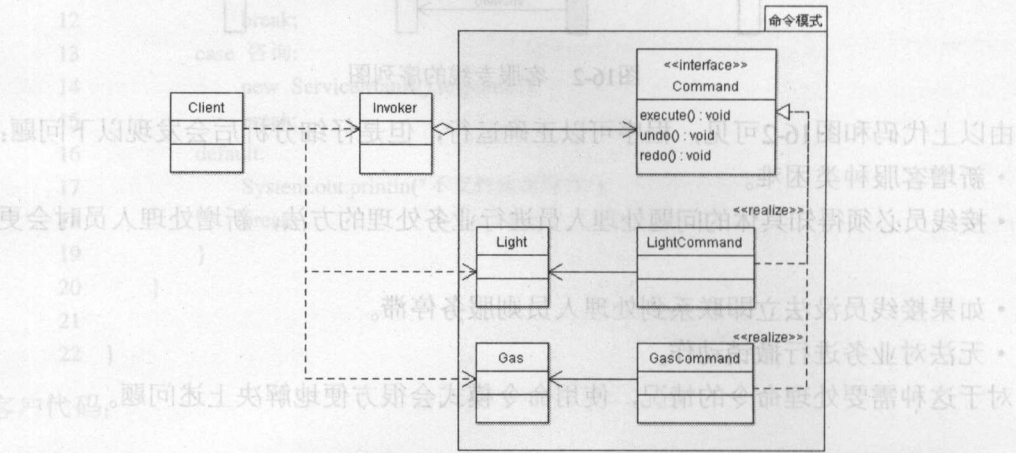


图16-5 示例类图

其中的参与者如下：

- Command接口（命令）：定义执行操作的接口。
- LightCommand、GasCommand(具体命令)：绑定Receiver对象和action、通过调用receiver的相应操作实现excute方法。
- Client（客户端，命令应用）：创建Concrete对象并设置其receiver。
- Invoker（调用者，使用者）：让命令执行请求。
- Light、Gas（即为Receiver（接收者））：执行与请求相应的操作。

首先看一下接收者的代码。

灯类：

代码片段6 Light.java

```
1 package cn.steven.pattern.demo.command.pattern;  
2  
3 /**  
4  * 接收者，灯
```

```

5  */
6  public class Light {
7
8      /**
9       * 灯的状态, true为开着, false为关闭
10      */
11     private boolean open = false;
12
13     public boolean isOpen() {
14         return open;
15     }
16
17     public void setOpen(boolean open) {
18         if (this.open == open) {
19             return;
20         } else {
21             this.open = open;
22             System.out.println("灯状态修改为: " + (open ? "打开": "关闭"));
23         }
24     }
25
26     /**
27      * 开灯方法
28      */
29     public void turnOnLight() {
30         this.setOpen(true);
31     }
32
33     /**
34      * 关灯方法
35      */
36     public void turnOffLight() {
37         this.setOpen(false);
38     }
39
40 }

```

煤气代码:

代码片段7 Gas.java

```

1  package cn.steven.pattern.demo.command.pattern;
2
3  /**
4   * 接收者, 煤气
5   */
6  public class Gas {
7
8      /**
9       * 煤气的状态, true为开着, false为关闭
10      */
11     private boolean fire = false;

```

```

12
13     public boolean isFire() {
14         return fire;
15     }
16
17     public void setFire(boolean fire) {
18         if (this.fire == fire) {
19             return;
20         } else {
21             this.fire = fire;
22             System.out.println("煤气状态修改为: " + (fire ? "打开" : "关闭"));
23         }
24     }
25
26     /**
27      * 打开煤气
28      */
29     public void openFire() {
30         this.setFire(true);
31     }
32
33     /**
34      * 关闭煤气
35      */
36     public void closeFire() {
37         this.setFire(false);
38     }
39 }

```

注意代码片段6和代码片段7非常相似，它们都有各自的操作方法，并且设置状态时使用了相应方法来判断是否需要真正修改状态。

命令接口：

代码片段8 Command.java

```

1 package cn.steven.pattern.demo.command.pattern;
2
3 /**
4  * 命令接口
5  */
6 public interface Command {
7
8     /**
9      * 执行方法
10     */
11     void execute();
12
13     /**
14      * 撤销方法
15     */
16     void undo();
17 }

```



```

18  /**
19   * 重做方法
20   */
21  void redo();
22  }

```

具体的命令类LightCommand:

代码片段9 LightCommand.java

```

1  package cn.steven.pattern.demo.command.pattern;
2
3  /**
4   * 具体命令
5   */
6  public class LightCommand implements Command {
7
8      /**
9       * 委托的接收者
10      */
11     private Light light;
12
13     /**
14      * 保存此动作是否成功修改的数据
15      */
16     private boolean change;
17
18     public Light getLight() {
19         return light;
20     }
21
22     public void setLight(Light light) {
23         this.light = light;
24     }
25
26     public boolean isChange() {
27         return change;
28     }
29
30     public void setChange(boolean change) {
31         this.change = change;
32     }
33
34     /**
35      * 构造方法
36      */
37     public LightCommand(Light light) {
38         this.setLight(light);
39         change = false;
40     }
41
42     @Override

```

```
43 public void execute() {
44     if (light.isOpen()) {
45         System.out.println("灯未做操作。");
46     } else {
47         light.turnOnLight();
48         System.out.println("灯已打开。");
49         change = true;
50     }
51 }
52
53 @Override
54 public void redo() {
55     execute();
56 }
57
58 @Override
59 public void undo() {
60     if (change) {
61         light.turnOffLight();
62         System.out.println("灯已关闭。");
63     } else {
64         System.out.println("灯未做操作。");
65     }
66 }
67
68 }
```

煤气命令类:

代码片段10 GasCommand.java

```
1 package cn.steven.pattern.demo.command.pattern;
2
3 /**
4  * 具体命令
5  */
6 public class GasCommand implements Command {
7
8     /**
9      * 委托的接收者
10     */
11     private Gas gas;
12
13     /**
14      * 保存此动作是否成功修改的数据
15     */
16     private boolean change;
17
18     public boolean isChange() {
19         return change;
20     }
21
22     @Override
23     public void execute() {
24         gas.turnOnLight();
25         change = true;
26     }
27
28     @Override
29     public void redo() {
30         execute();
31     }
32
33     @Override
34     public void undo() {
35         gas.turnOffLight();
36         change = false;
37     }
38 }
```

```

22 public void setChange(boolean change) {
23     this.change = change;
24 }
25
26 public Gas getGas() {
27     return gas;
28 }
29
30 public void setGas(Gas gas) {
31     this.gas = gas;
32 }
33
34 /**
35  * 构造方法
36  *
37  * @param gas
38  */
39 public GasCommand(Gas gas) {
40     this.setGas(gas);
41     change = false;
42 }
43
44 @Override
45 public void execute() {
46     if(gas.isFire()){
47         System.out.println("煤气未做操作。(execute())");
48     }else{
49         gas.openFire();
50         System.out.println("煤气已打开。");
51         change = true;
52     }
53 }
54
55 @Override
56 public void redo() {
57     execute();
58 }
59
60 @Override
61 public void undo() {
62     if(change){
63         gas.closeFire();
64         System.out.println("煤气已关闭。");
65     }else{
66         System.out.println("煤气未做操作。(undo())");
67     }
68 }
69
70 }

```

调用命令类:

代码片段11 Invoker.java

```

1 package cn.steven.pattern.demo.command.pattern;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 /**
7  * 命令调用者
8  */
9 public class Invoker {
10
11     /**
12      * 保存命令的集合
13      */
14     private List<Command> list = new LinkedList<Command>();
15
16     public List<Command> getList() {
17         return list;
18     }
19
20     public void setList(List<Command> list) {
21         this.list = list;
22     }
23
24     /**
25      * 保存当前命令集合中的位置
26      */
27     private int position = -1;
28
29     /**
30      * 执行命令
31      *
32      * @param command
33      */
34     public void executeCommand(Command command) {
35         /**
36          * 加入命令队列
37          */
38         list.add(++position, command);
39
40         /**
41          * 执行命令
42          */
43         command.execute();
44
45         /**
46          * 将集合尾部的多余命令删除
47          */
48         if (position < this.getList().size() - 1) {
49             for (int i = this.getList().size() - 1; i > position; i--) {
50                 this.getList().remove(i);

```

```

51 已经显示 } 命令已经成功执行, 撤销和重做动作也是正确的。最后一行的命令
52  }
53  }
54  }
55  /**
56   * 重做命令
57   */
58  public void redoCommand() {
59      if (this.getList().size() - 1 > position) {
60          /**
61           * 执行命令
62           */
63          this.getList().get(++position).redo();
64      } else {
65          System.out.println("命令无效");
66      }
67  }
68
69  /**
70   * 撤销命令
71   */
72  public void undoCommand() {
73      if (position >= 0) {
74          /**
75           * 撤销命令
76           */
77          this.getList().get(position--).undo();
78      }
79  }
80  }

```

注意代码片段11中使用了一个List来保存所执行过的命令, 这样就对撤销和重做命令提供了支持。通常这些约定为常用的特征: 撤销和恢复的约定是如果撤销命令后又做了新的非撤销和恢复的命令动作, 则这个命令动作将替代可供重做的所有命令, 命令链被截断了。代码中第45行~第53行即是执行这段特征的代码。

客户端代码:

代码片段12 Client.java

```

1  package cn.steven.pattern.demo.command.pattern;
2
3  /**
4   * 客户端代码
5   */
6  public class Client {
7      public static void main(String[] args) {
8          /**
9           * 需要操作的物品
10          */
11          Light light = new Light();
12          Gas gas = new Gas();

```

```
13
14  /**
15  package cn. * 创建调用者
16  */
17  import java. Invoker invoker = new Invoker();
18  import java.util.List;
19  /**
20  * 执行命令
21  */
22  invoker.executeCommand(new GasCommand(gas));
23  public class invoker.executeCommand(new GasCommand(gas));
24
25  /**
26  invoker.undoCommand();
27  invoker.undoCommand();
28  invoker.redoCommand();
29
30  invoker.executeCommand(new LightCommand(light));
31
32  invoker.redoCommand();
33
34  }
35  }
```

运行时的序列图如图16-6所示。

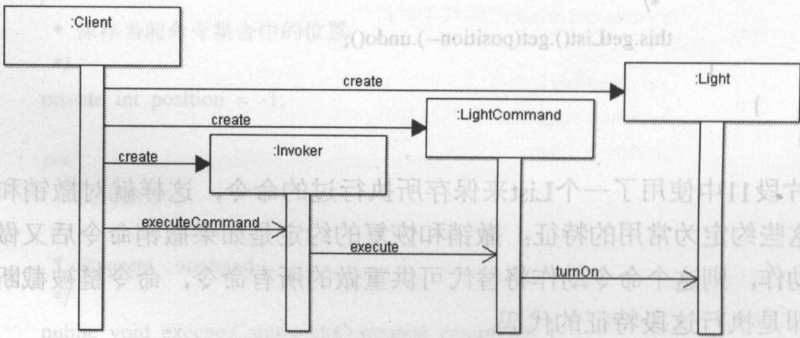


图16-6 命令模式序列图

命令执行结果如图16-7所示。

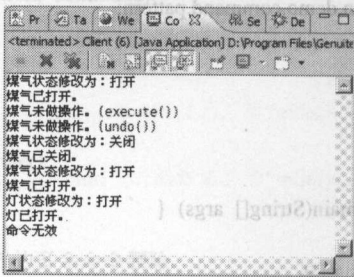


图16-7 运行结果

注意结果已经显示各个命令已经成功执行，撤销和重做动作也是正确的。最后一行的命令无效是因为命令链条被截断，所以最后的一个命令被删除。

16.2.2 使用命令模式解决客服电话的问题

经过了上一节的学习，下面已经可以使用命令模式来解决客服专线的问题了，此解决方法的重点之处就在于：

- 封装命令，对外界提供命令接口。
- 命令具备执行、撤销和重做功能。
- 使用调用者对象来执行所有命令。
- 具体命令委托外部的类执行业务操作。

现在除了以上所述功能外，还增加了一个批处理的功能，此功能的作用是使用一个命令对象来代替多个命令对象，具体的实现方法就是前面所学的一种结构模式——组合模式。设计类图如图16-8所示。

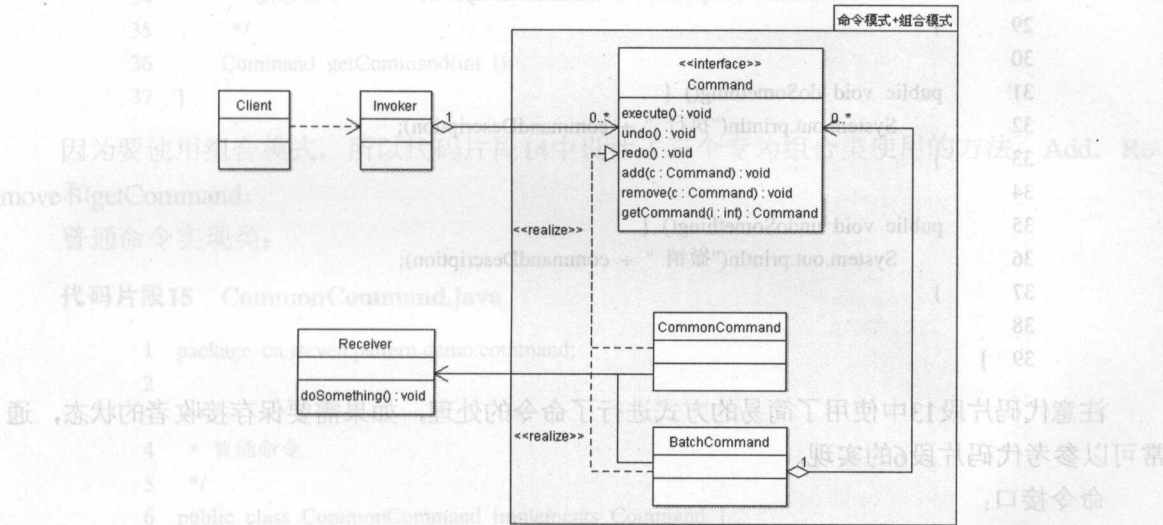


图16-8 客服专线问题的设计类图

为了简化设计，突出重点部分，这里的接收者被设计为一个简化的模型，此设计中最复杂的实现部分就是调用者对于批处理命令的处理过程，读者要重点观察其撤销命令的实现。

接收者代码如下：

代码片段13 Receiver.java

```
1 package cn.steven.pattern.demo.command;
2
3 /**
4  * 命令接收者
5  */
6 public class Receiver {
7
8     /**
9      * 命令描述
10     */
```

```

11 private String commandDescription;
12
13 public String getCommandDescription() {
14     return commandDescription;
15 }
16
17 public void setCommandDescription(String commandDescription) {
18     this.commandDescription = commandDescription;
19 }
20
21 /**
22  * 构造方法
23  *
24  * @param commandDescription
25  */
26 public Receiver(String commandDescription) {
27     super();
28     this.commandDescription = commandDescription;
29 }
30
31 public void doSomething() {
32     System.out.println("执行 " + commandDescription);
33 }
34
35 public void undoSomething() {
36     System.out.println("撤销 " + commandDescription);
37 }
38
39 }

```

注意代码片段13中使用了简易的方式进行了命令的处理，如果需要保存接收者的状态，通常可以参考代码片段6的实现。

命令接口：

代码片段14 Command.java

```

1 package cn.steven.pattern.demo.command;
2
3 /**
4  * 命令接口
5  */
6 public interface Command {
7
8     /**
9      * 执行方法
10     */
11     void execute();
12
13     /**
14      * 撤销方法
15     */
16     void undo();

```

```

17 public class BatchCommand implements Command {
18     /**
19      * 重做方法
20      */
21     void redo();
22
23     /**
24      * 增加命令
25      */
26     void add(Command c);
27
28     /**
29      * 删除命令
30      */
31     void remove(Command c);
32
33     /**
34      * 获取命令
35      */
36     Command getCommand(int i);
37 }

```

因为要使用组合模式，所以代码片段14中设计了三个专为组合类使用的方法：Add、Remove和getCommand。

普通命令实现类：

代码片段15 CommonCommand.java

```

1 package cn.steven.pattern.demo.command;
2
3 /**
4  * 普通命令
5  */
6 public class CommonCommand implements Command {
7
8     /**
9      * 接收者
10     */
11     private Receiver receiver;
12
13     public Receiver getReceiver() {
14         return receiver;
15     }
16
17     public void setReceiver(Receiver receiver) {
18         this.receiver = receiver;
19     }
20
21     /**
22      * 构造方法
23      *
24      * @param receiver

```



```

25     */
26     public CommonCommand(Receiver receiver) {
27         super();
28         this.receiver = receiver;
29     }
30
31     @Override
32     public void add(Command c) {
33         throw new RuntimeException("不支持此方法！");
34     }
35
36     @Override
37     public void execute() {
38         receiver.doSomething();
39     }
40
41     @Override
42     public Command getCommand(int i) {
43         throw new RuntimeException("不支持此方法！");
44     }
45
46     @Override
47     public void redo() {
48         receiver.doSomething();
49     }
50
51     @Override
52     public void remove(Command c) {
53         throw new RuntimeException("不支持此方法！");
54     }
55
56     @Override
57     public void undo() {
58         receiver.undoSomething();
59     }
60
61 }

```

很显然代码片段15并不是一个组合类，所以接口中的组合类方法在这里采用抛出运行时异常的方法来处理。

组合命令类：

代码片段16 BatchCommand.java

```

1 package cn.steven.pattern.demo.command;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 /**
7  * 批处理命令
8  */

```

```

9 public class BatchCommand implements Command {
10
11     /**
12      * 命令列表
13      */
14     private List<Command> commandList;
15
16     public List<Command> getCommandList() {
17         return commandList;
18     }
19
20     public void setCommandList(List<Command> commandList) {
21         this.commandList = commandList;
22     }
23
24     /**
25      * 构造方法
26      */
27     public BatchCommand() {
28         commandList = new LinkedList<Command>();
29     }
30
31     @Override
32     public void add(Command c) {
33         commandList.add(c);
34     }
35
36     @Override
37     public void execute() {
38         for (Command c : commandList) {
39             c.execute();
40         }
41     }
42
43     @Override
44     public Command getCommand(int i) {
45         return commandList.get(i);
46     }
47
48     @Override
49     public void redo() {
50         for (Command c : commandList) {
51             c.redo();
52         }
53     }
54
55     @Override
56     public void remove(Command c) {
57         commandList.remove(c);
58     }
59

```

```

60     @Override
61     public void undo() {
62         /**
63          * 注意此处撤销需要按倒序执行
64          */
65         for (int i = commandList.size() - 1; i >= 0; i--) {
66             commandList.get(i).undo();
67         }
68     }
69 }
70 }

```

请重点关注代码片段16中第60行~第68行的撤销方法，按照语意的合理性，此处的撤销必须严格按倒序执行。

命令调用者类：

代码片段17 Invoker.java

```

1  package cn.steven.pattern.demo.command;
2
3  import java.util.LinkedList;
4  import java.util.List;
5
6  /**
7   * 命令调用者
8   */
9  public class Invoker {
10
11     /**
12      * 保存命令的集合
13      */
14     private List<Command> list = new LinkedList<Command>();
15
16     public List<Command> getList() {
17         return list;
18     }
19
20     public void setList(List<Command> list) {
21         this.list = list;
22     }
23
24     /**
25      * 保存当前命令集合中的位置
26      */
27     private int position = -1;
28
29     /**
30      * 执行命令
31      *
32      * @param command
33      */
34     public void executeCommand(Command command) {

```



```
35  /**
36   * 加入命令队列
37   */
38   list.add(++position, command);
39
40  /**
41   * 执行命令
42   */
43   command.execute();
44
45  /**
46   * 将集合尾部的多余命令删除
47   */
48   if (position < this.getList().size() - 1) {
49       for (int i = this.getList().size() - 1; i > position; i--) {
50           this.getList().remove(i);
51       }
52   }
53
54  /**
55   * 重做命令
56   */
57
58   public void redoCommand() {
59       if (this.getList().size() - 1 > position) {
60           /**
61            * 执行命令
62            */
63           this.getList().get(++position).redo();
64       } else {
65           System.out.println("命令无效");
66       }
67   }
68
69  /**
70   * 撤销命令
71   */
72   public void undoCommand() {
73       if (position >= 0) {
74           /**
75            * 撤销命令
76            */
77           this.getList().get(position--).undo();
78       }
79   }
80 }
```

客户端代码:

代码片段18 Client.java

```
1 package cn.steven.pattern.demo.command;
```

```

2
3 /**
4  * 客户端
5  */
6 public class Client {
7     public static void main(String[] args) {
8
9         /**
10          * 创建调用者
11          */
12         Invoker invoker = new Invoker();
13
14         /**
15          * 执行普通命令
16          */
17         invoker.executeCommand(new CommonCommand(
18             new Receiver("询问价格")));
19         invoker.executeCommand(new CommonCommand(
20             new Receiver("投诉商品")));
21
22         invoker.undoCommand();
23
24         invoker.redoCommand();
25
26         /**
27          * 创建批处理命令
28          */
29         BatchCommand bc = new BatchCommand();
30         bc.add(new CommonCommand(new Receiver("询问电视价格")));
31         bc.add(new CommonCommand(new Receiver("购买电视")));
32         bc.add(new CommonCommand(new Receiver("电视送货")));
33
34         invoker.executeCommand(bc);
35
36         invoker.undoCommand();
37     }
38 }

```

请注意代码片段18中命令调用的先后顺序及撤销和重做的顺序。
执行结果如图16-9所示。

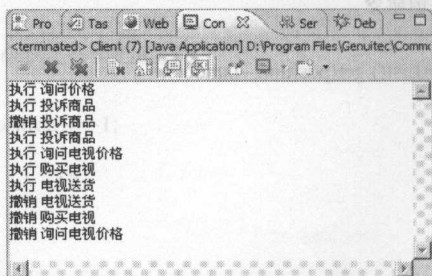


图16-9 Client.java执行结果


```
22     HttpServletResponse response)
23     throws Exception {
24     return null;
25     }
26 }
```

请求者RequestProcessor:

代码片段20 RequestProcessor.java

```
1 public class RequestProcessor {
2     protected ActionForward processActionPerform(
3         HttpServletRequest request,
4         HttpServletResponse response,
5         Action action,
6         ActionForm form,
7         ActionMapping mapping)
8     throws IOException, ServletException {
9     try {
10         return (action.execute(mapping, form,
11             request, response));
12     } catch (Exception e) {
13         return (processException(request, response,
14             e, form, mapping));
15     }
16 }
17 }
```

Struts框架为我们提供了以上的两个角色，要使用Struts框架完成自己的业务逻辑，剩下的三个角色就要由我们自己来实现了。步骤如下：

- 实现一个Action的子类，并重写execute方法。在此方法中调用业务模块的相应对象来完成任务。
- 实现处理业务的业务类。
- 配置struts-config.xml配置文件，将自己的Action和Form以及相应页面结合起来。
- 编写JSP脚本，在页面中显式制定对应的处理Action。

当在页面上提交请求后，Struts框架会根据配置文件中的定义，将你的Action对象作为参数传递给RequestProcessor类中的processActionPerform()方法，由此方法调用Action对象中的执行方法，进而调用业务层中的接收角色。这样就完成了请求的处理。

由以上例子可见，命令模式的封装命令的作用在很多场合下都会用到，读者可以自行总结各种框架中此模式的使用代码。

16.2.4 命令模式的使用范围

命令模式的优点：

- 很容易构造一个命令队列。
- 记录相关的命令日志。
- 增加命令的状态，实现命令的撤销和重做。
- 允许接收请求的一方决定是否可做。

- 新的命令可以轻易加入其中。

命令模式的缺点:

- 可能会有过多的具体命令类存在。

适用场合:

• 抽象出待执行的动作以参数化某对象时。可用过程语言中的回调 (Call back) 函数表达这种参数化机制。所谓回调函数是指函数先在某处注册, 而在稍后某个需要的时候它将被调用。Command模式是回调机制的一个面向对象的替代品。

• 要在不同的时刻指定、排列和执行请求时。一个Command对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达, 那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。

• 要支持取消操作时。Command的execute操作可在实施操作前将状态存储起来, 在执行取消操作时, 这个状态用来消除该操作的影响。Command接口必须添加一个undo操作, 该操作取消上一次execute调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用 undo和redo来实现重数不限的“撤销”和“重做”操作。

• 要支持修改日志时, 这样当系统崩溃时, 这些修改可以被重做一遍。在Command接口中添加装载操作和存储操作, 可以用来保持修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用execute操作重新执行它们。

• 用构建在原语操作上的高层操作构造一个系统时。这样一种结构在支持事务 (transaction) 的信息系统中很常见。一个事务封装了对数据的一组变动。命令模式可提供对事务进行建模的方法。Command有一个公共的接口, 使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

16.2.5 与其他模式的关系

命令模式和组合模式经常在一起使用, 用以实现批处理命令 (宏命令) 的效果。

命令模式可以和备忘录模式一起使用, 作用是在撤销和恢复命令时保存其状态。

命令有时候需要被复制, 可以使用原型模式。

命令模式和策略模式辨析:

策略模式把易于变化的行为分别封装起来, 让它们之间可以互相替换, 让这些行为的变化独立于拥有这些行为的客户。

命令模式是一种对象行为型模式, 它主要解决的问题是: 在软件构建过程中, “行为请求者”与“行为实现者”通常呈现一种“紧耦合”的问题。

16.3 命令模式总结

命令模式解决了“行为请求者”与“行为实现者”之间紧密耦合的问题, 并且日志记录、命令撤销、批处理、事务支持也是它能带来的好处, 读者在使用命令模式时通常要和其他模式相结合使用, 比如组合模式、备忘录模式等。

在学习的时候最好找一些JDK或Java类库中的相关代码辅助阅读, 这样会起到更好的效果。

第17章 观察者模式 (Observer)

观察者模式属于行为型模式，其意图是定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新¹。

在日常生活中，有时有这样的需求，就是有很多人需要得到某种消息，而且是最新的消息，这样的需求体现在很多方面，比如每天的报纸，新的报纸印刷好后订报的人就需要得到消息。再比如很多人都买了“中国石油”的股票，当这只股票的价格变动的时候，这些人肯定都想立刻得到新的股价。

在企业中也有这样类似的需求，如图17-1所示。

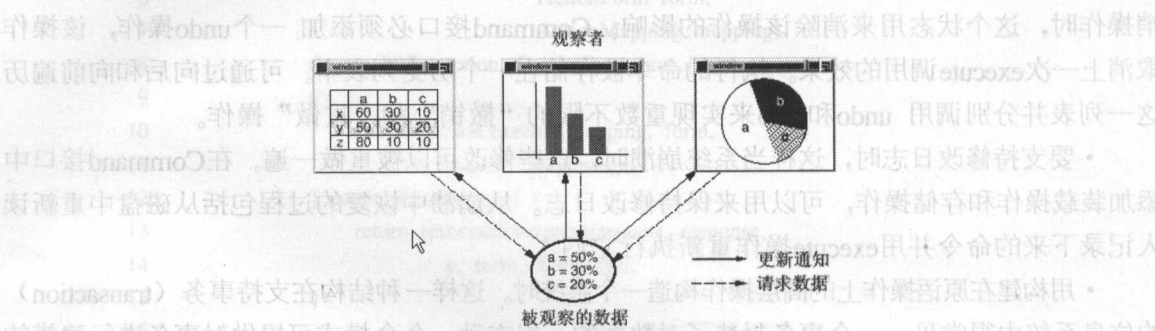


图17-1 报表例子

图17-1中所展示的是一个假想的企业报表案例，比如公司中有几类不同的人需要看到公司报表不同的表现形式（视图），如财务人员看表格形式的，总经理看柱状图，而董事会要看饼状图，这些视图所反映的数据应该是一样的，当数据改动时，这些种类的视图应该同步变化。

像以上各个例子的需求就是典型的数据和展示相分离（观察对象和被观察的主题对象相分离）的案例，此类问题解决的最好方式就是使用观察者模式。

17.1 连锁店宣传资料的发放问题

连锁店总店有一个宣传部，这个部门的任务是不定期地发放连锁店的各种消息，比如打折促销、品牌宣传、有奖销售等的活动信息，一旦有新的消息发出，通常又会通过电视广告、特定客户发放、街头宣传单、户外宣传海报等的形式进行宣传。现在的问题就是解决及时通知最新消息到各种媒体的事情。

像这样的消息通知功能，通常有两种解决方案。

一种是客户端询问实现，典型的案例是老式的自动刷新聊天室，聊天室里有多用户同时在聊天，每个客户都需要知道当前聊天室的记录，采用的办法就是每隔一段时间（比如3秒钟），就重新请求一次聊天室程序来得到最新的聊天记录。这种实现方式的缺点是服务器的压

¹Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.GOF[95]

力很大，客户端数量的支持有限。

另一种方式是服务器端发送通知，它比客户端询问这种方案的效率高，客户端支持的数量也有提高，这种方案又有两个具体的实现方案：

- 推模式是当有消息时，把消息信息以参数的形式传递（推）给所有观察者。
- 拉模式是当有消息时，通知消息的方法本身并不带任何的参数，是由观察者自己到主体对象那儿取回（拉）消息。

报纸订阅过程的举例如下：

在推环境中，发行方很可能会雇佣投送人员四处送报。换句话说，他们把自己的报纸推出去，让订阅者收取。在拉环境中，规模较小的本地报社可能会在订阅者家附近的街角提供自己的报纸，供订阅者“拉”。那些成长型发行方没有足够的资源进行大规模投送，因此一般采用拉方案，让订阅者到当地的杂货店或自动售货机那里“拿”报，这种方案对于他们来说往往是个优化投送环节的好办法。

下面就是根据需求做出的初步解决方案类图，如图17-2所示。

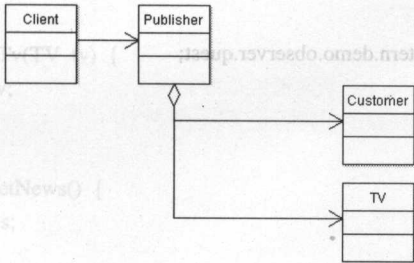


图17-2 初步设计类图

- 序列图如图17-3所示。

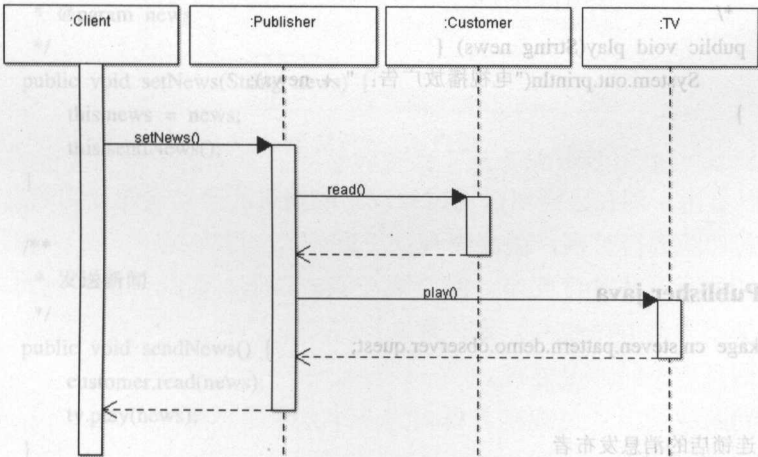


图17-3 初步设计序列图

下面是两个需要得到最新消息的类：

顾客类：

代码片段1 Customer.java

```
1 package cn.steven.pattern.demo.observer.quest;
```

```
2
3 /**
4  * 顾客
5  */
6 public class Customer {
7
8     /**
9      * 阅读消息
10     *
11     * @param news
12     */
13     public void read(String news) {
14         System.out.println("顾客阅读了: " + news);
15     }
16 }
```

电视类:

代码片段2 TV.java

```
1 package cn.steven.pattern.demo.observer.quest;
2
3 /**
4  * 电视节目
5  */
6 public class TV {
7
8     /**
9      * 广播消息
10     *
11     * @param news
12     */
13     public void play(String news) {
14         System.out.println("电视播放广告: " + news);
15     }
16
17 }
```

发布者类:

代码片段3 Publisher.java

```
1 package cn.steven.pattern.demo.observer.quest;
2
3 /**
4  * 连锁店的信息发布者
5  */
6 public class Publisher {
7
8     /**
9      * 消息内容
10     */
11     private String news;
```

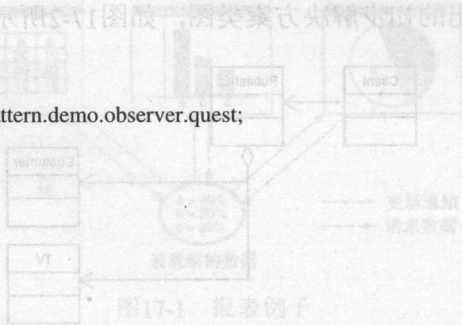


图 17-1 报告例子

17.1 连锁店宣传资料的发放问题

有一个宣传部，这个部门负责管理连锁店的各种信息，比如打折促销、新品上市、会员活动等。一旦有这些信息，宣传部就会通过电视广告、特定客户广告等方式，将这些信息传达给连锁店。这就是解决及时通知最新消息到连锁店的问题。

通常有两种解决方案。

一种方案是，每个连锁店都有一个用户，这个用户会定期（比如3秒钟）向宣传部请求最新的消息。这种方案的问题是，如果连锁店很多，那么宣传部就需要处理大量的请求，这会导致性能下降。

另一种方案是，宣传部维护一个消息列表，每个连锁店都有一个用户，这个用户会定期向宣传部请求最新的消息。这种方案的问题是，如果连锁店很多，那么宣传部就需要维护大量的消息列表，这也会导致性能下降。

为了解决这个问题，我们可以使用观察者模式。观察者模式是一种设计模式，它允许一个对象（发布者）维护一个观察者列表，当发布者状态发生变化时，它会通知所有观察者，让观察者更新它们的状态。这种模式可以有效地解决及时通知的问题，因为它可以避免大量的请求，并且可以确保所有观察者都能及时收到最新的消息。

```

12
13 /**
14  * 消息接收者
15  */
16 private Customer customer;
17 private TV tv;
18
19 public Customer getCustomer() {
20     return customer;
21 }
22
23 public void setCustomer(Customer customer) {
24     this.customer = customer;
25 }
26
27 public TV getTv() {
28     return tv;
29 }
30
31 public void setTv(TV tv) {
32     this.tv = tv;
33 }
34
35 public String getNews() {
36     return news;
37 }
38
39 /**
40  * 设置方法说明消息已经修改
41  *
42  * @param news
43  */
44 public void setNews(String news) {
45     this.news = news;
46     this.sendNews();
47 }
48
49 /**
50  * 发送新闻
51  */
52 public void sendNews() {
53     customer.read(news);
54     tv.play(news);
55 }
56
57 }

```

客户端类:

代码片段4 Client.java

```
1 package cn.steven.pattern.demo.observer.quest;
```



```
2
3 /**
4  * 运行代码
5  */
6 public class Client {
7     public static void main(String[] args) {
8         /**
9          * 创建出版者
10         */
11         Publisher publisher = new Publisher();
12         /**
13          * 设置接收者
14         */
15         publisher.setCustomer(new Customer());
16
17         publisher.setTv(new TV());
18
19         /**
20          * 公布消息
21         */
22         publisher.setNews("新连锁店开张了。");
23         publisher.setNews("打折促销了。");
24     }
25 }
```

运行结果如图17-4所示。

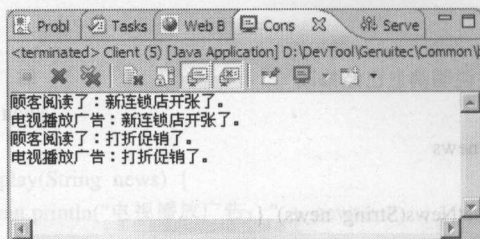


图17-4 运行结果

以上结果可以看出，现在已经可以通过修改新的消息而自动通知订阅消息的对象更新了。但是，如果想对其功能进行扩展，就会发现如下问题：

- 发布者维护的是具体的订阅者，两者耦合紧密，当有新的订阅者类时，扩充很困难。
- 订阅者的调用方法并不统一，操作复杂。
- 当传递的信息复杂时不易处理。

以上类型的问题就可以通过观察者模式的设计方法得到解决。

17.2 观察者模式的结构

观察者模式的精华部分是对主题对象和订阅对象的联系解耦合。

17.2.1 观察者模式

观察者模式是一步一步演化而来的。这里使用消息订阅作为例子看一下其演化过程，学习其演化过程对于程序员来说可以锻炼其抽象的能力。

最初，当主题方的信息发生了变化，它将直接通知其内部聚合的观察者对象，其类关系如图17-5所示。

注意由于主题直接耦合异变的观察者方，因此此处需要重构，让主题方依赖不异变的接口，如图17-6所示。

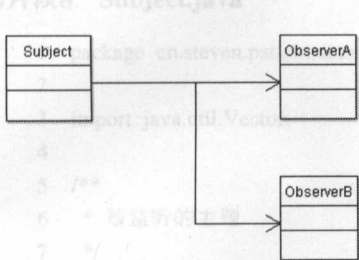


图17-5 观察者模式演化-1

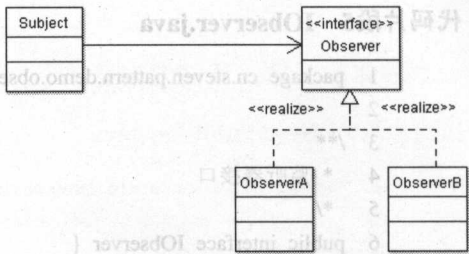


图17-6 观察者模式演化-2

现在观察者的易变问题就解决了，但是我们可以发现主题方不具备复用性，所以继续演化如下，如图17-7所示。

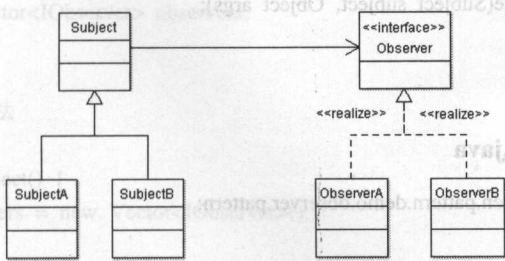


图17-7 观察者模式演化-3

经过上述设计过程，对于出版和订阅过程有以下最终的设计版本，如图17-8所示。

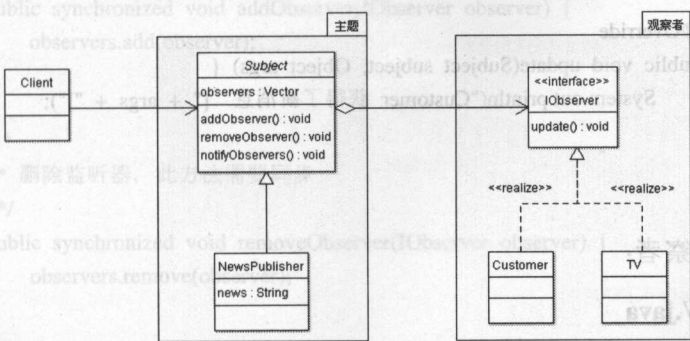


图17-8 观察者模式演化-4

图17-8中的参与者如下：

- Subject（被观察的对象接口）：规定ConcreteSubject的统一接口。每个Subject可以有多

个IObserver。

- **NewsPublisher**（具体被观察对象）：维护对所有具体观察者的引用的列表。状态发生变化时会发送通知给所有注册的观察者。
- **IObserver**（观察者接口）：规定ConcreteObserver的统一接口。定义了一个update()方法，在被观察对象状态改变时会被调用。
- **Customer、TV**（具体观察者）：维护一个对具体被观察对象的引用。特定状态与具体被观察对象同步。实现IObserver接口，通过update()方法接收具体被观察对象的通知。

下面展示具体代码，首先是高层的观察者接口：

代码片段5 IObserver.java

```
1 package cn.steven.pattern.demo.observer.pattern;
2
3 /**
4  * 监听器接口
5  */
6 public interface IObserver {
7     /**
8      * 更新方法
9      * @param subject 主题对象
10     * @param args 更新参数
11     */
12     void update(Object subject, Object args);
13 }
```

一个具体的观察者：

代码片段6 Customer.java

```
1 package cn.steven.pattern.demo.observer.pattern;
2
3 /**
4  * 顾客类
5  */
6 public class Customer implements IObserver {
7
8     @Override
9     public void update(Object subject, Object args) {
10         System.out.println("Customer 获得了新消息 [" + args + "]");
11     }
12 }
13 }
```

另一个具体观察者：

代码片段7 TV.java

```
1 package cn.steven.pattern.demo.observer.pattern;
2
3 /**
4  * 电视类
5  */
```



```

6 public class TV implements IObserver {
7     public static void main(String[] args) {
8         @Override
9         public void update(Subject subject, Object args) {
10             System.out.println("TV 获得了新消息 [" + args + "]");
11         }
12     }
13 }

```

主题抽象类:

代码片段8 Subject.java

```

1 package cn.steven.pattern.demo.observer.pattern;
2
3 import java.util.Vector;
4
5 /**
6  * 被监听的主题
7  */
8 public abstract class Subject {
9
10     /**
11      * 用于存放监听器的集合，此集合线程安全
12      */
13     private Vector<IObserver> observers;
14
15     /**
16      * 构造方法
17      */
18     public Subject() {
19         observers = new Vector<IObserver>();
20     }
21
22     /**
23      * 新增监听器，此方法需要同步
24      */
25     public synchronized void addObserver(IObserver observer) {
26         observers.add(observer);
27     }
28
29     /**
30      * 删除监听器，此方法需要同步
31      */
32     public synchronized void removeObserver(IObserver observer) {
33         observers.remove(observer);
34     }
35
36     /**
37      * 删除所有监听器，此方法需要同步
38      */
39     public synchronized void removeObservers() {

```

```
40         observers.clear();
41     }
42
43     /**
44      * 通知方法
45      */
46     public synchronized void notifyObservers(Object args) {
47         for (IObserver observer : observers) {
48             observer.update(this, args);
49         }
50     }
51 }
```

一个具体的主题：新闻发布者。

代码片段9 NewsPublisher.java

```
1 package cn.steven.pattern.demo.observer.pattern;
2
3 /**
4  * 新闻发布类
5  */
6 public class NewsPublisher extends Subject {
7
8     /**
9      * 维护着最新的信息
10    */
11    private String news;
12
13    public void setNews(String news) {
14        /**
15         * 消息改变时通知
16         */
17        if (news != null && !news.equals(this.news)) {
18            this.news = news;
19            /**
20             * 通知监听器
21             */
22            this.notifyObservers(news);
23        }
24    }
25
26 }
```

客户端代码如下：

代码片段10 Client.java

```
1 package cn.steven.pattern.demo.observer.pattern;
2
3 /**
4  * 客户代码
5  */
```

```
6 public class Client {
7     public static void main(String[] args) {
8         /**
9          * 创建主题类
10         */
11         NewsPublisher newsPublisher = new NewsPublisher();
12
13         /**
14          * 增加监听器
15         */
16         newsPublisher.addObserver(new Customer());
17         newsPublisher.addObserver(new TV());
18
19         /**
20          * 改变主题内容
21         */
22         newsPublisher.setNews("空调大降价");
23     }
24 }
25
26 }
```

运行结果如图17-9所示。

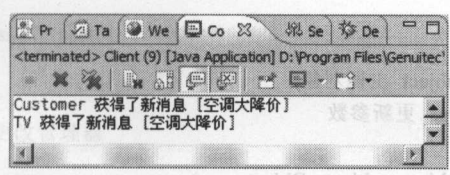


图17-9 运行结果

如图17-9所示即为观察者模式的运行结果，此模式可以通过一个规则的方式使观察者和主题之间的联系解耦合。但是，以上例子只使用了一种消息更新（新闻内容）和一个主题对象（新闻发布者）。

在有多种消息订阅的情况下，建议使用JDK提供的“事件/监听器”机制。在有多种主题对象需要订阅时，建议使用“更改管理器”机制。

17.2.2 使用观察者模式构建发布/订阅模型

经过了上一节的学习，下面已经可以使用观察者模式解决宣传单的发布/订阅问题了，但是接下来的设计要比原始的观察者模式更复杂，其中增加了：

- 多个主题。
- 多个观察者。
- 更改管理器单例类。
- 主题和观察者之间是多对多的关系。

更改管理器的引入是为了解决多个主题的订阅管理分散的问题，其类图如图17-10所示。

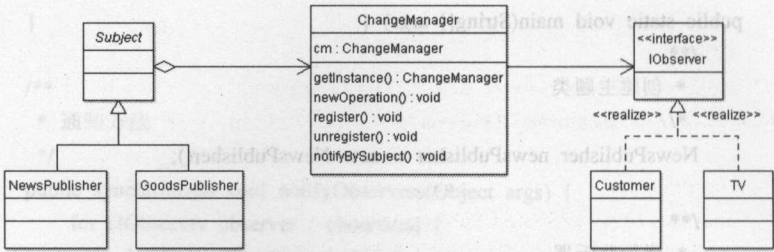


图17-10 更改管理器观察者模式

由图17-10可见，加入了更改管理器之后，主题和观察者之间的耦合度更加弱了，甚至连主题和观察者之间的映射管理也被独立了出来，下面我们就来看一下实现代码。

首先看一下观察者的具体代码：

代码片段11 IObserver.java

```
1 package cn.steven.pattern.demo.observer;
2
3 /**
4  * 监听器接口
5  */
6 public interface IObserver {
7     /**
8      * 更新方法
9      * @param subject 主题对象
10     * @param args 更新参数
11     */
12     void update(Subject subject, Object args);
13 }
```

具体实现类1:

代码片段12 Customer.java

```
1 package cn.steven.pattern.demo.observer;
2
3 /**
4  * 顾客类
5  */
6 public class Customer implements IObserver {
7
8     @Override
9     public void update(Subject subject, Object args) {
10         System.out.println("Customer 获得了新消息 [" + args + "]");
11     }
12
13 }
```

具体实现类2:

代码片段13 TV.java

```
1 package cn.steven.pattern.demo.observer;
```

```

2  * 商品发布类
3  /**
4  * 电视类
5  */
6  public class TV implements IObserver {
7
8      @Override
9      public void update(Subject subject, Object args) {
10         System.out.println("TV 获得了新消息 [" + args + "]");
11     }
12
13 }

```

由以上三段代码可见，观察者的实现和标准观察者模式是相同的，但是由于更改管理器的引入，主题类变得不同了。

抽象主题类：

代码片段14 Subject.java

```

1  package cn.steven.pattern.demo.observer;
2
3  /**
4  * 抽象主题类
5  */
6  public abstract class Subject {
7
8      /**
9       * 保存一个更改管理器
10     */
11     private static final ChangeManager changeManager = ChangeManager
12         .getInstance();
13
14     public static ChangeManager getChangeManager() {
15         return changeManager;
16     }
17
18     /**
19      * 增加监听器
20      *
21      * @param observer
22      */
23     public void addObserver(IObserver observer) {
24         getChangeManager().register(this, observer);
25     }
26
27     /**
28      * 移除监听器
29      *
30      * @param observer
31      */
32     public void removeObserver(IObserver observer) {
33         getChangeManager().unregister(this, observer);

```

```
34     }
35
36     /**
37      * 通知更新
38      *
39      * @param args
40      */
41     public void notify(Object args) {
42         getChangeManager().notifyBySubject(this, args);
43     }
44
45 }
```

图17-10 更改管理器观察者模式

由代码片段14可见，管理和运行观察者对象的功能已经委托给ChangeManager对象了。ChangeManager就是一个更改管理类。

具体新闻主题类：

代码片段15 NewsPublisher.java

```
1 package cn.steven.pattern.demo.observer;
2
3 /**
4  * 新闻发布类
5  */
6 public class NewsPublisher extends Subject {
7
8     /**
9      * 维护着最新的信息
10     */
11     private String news;
12
13     public void setNews(String news) {
14         /**
15          * 消息改变时通知
16          */
17         if (news != null && !news.equals(this.news)) {
18             this.news = news;
19             /**
20              * 通知监听器
21              */
22             this.notify(news);
23         }
24     }
25
26 }
```

具体商品主题类：

代码片段16 GoodsPublisher.java

```
1 package cn.steven.pattern.demo.observer;
2
3 /**
```



```

4  * 商品发布类
5  */
6  public class GoodsPublisher extends Subject {
7
8      /**
9       * 维护着最新的信息
10     */
11     private String news;
12
13     public void setNews(String news) {
14         /**
15          * 消息改变时通知
16          */
17         if (news != null && !news.equals(this.news)) {
18             this.news = news;
19             /**
20              * 通知监听器
21              */
22             this.notify(news);
23         }
24     }
25
26 }

```

下面就是核心的更改管理器类，注意此类实现为单例模式：

代码片段17 ChangeManager.java

```

1  package cn.steven.pattern.demo.observer;
2
3  import java.util.Hashtable;
4  import java.util.Vector;
5
6  /**
7   * 更改管理器（单例）
8   */
9  public class ChangeManager {
10
11     /**
12      * 单例属性
13      */
14     private static final ChangeManager cm = new ChangeManager();
15
16     /**
17      * 保存映射关系
18      */
19     private static Hashtable<Subject, Vector<IObserver>> map
20         = new Hashtable<Subject, Vector<IObserver>>();
21
22     /**
23      * 私有构造方法
24      */

```

```

25     private ChangeManager() {
26
27     }
28
29     /**
30      * 单例获得方法
31      */
32     public static ChangeManager getInstance(){
33         return cm;
34     }
35
36     /**
37      * 注册监听器
38      *
39      * @param subject
40      * @param observer
41      */
42     public synchronized void register(Subject subject,
43         IObservable observer) {
44
45         Subject key = subject;
46         Vector<IObservable> observers = null;
47
48         /**
49          * 查看是否已经注册此对象
50          */
51         if (map.containsKey(key)) {
52             observers = map.get(key);
53         }
54
55         /**
56          * 查看此对象对应的监听器列表是否存在
57          */
58         if (observers == null) {
59             observers = new Vector<IObservable>();
60         }
61
62         /**
63          * 查看监听器是否重复
64          */
65         if (!observers.contains(observer)) {
66             observers.add(observer);
67         }
68
69         /**
70          * 更新映射关系
71          */
72         map.put(key, observers);
73     }
74
75     /**

```

```

76  * 解除监听器注册
77  *
78  * @param subject
79  * @param observer
80  */
81  public synchronized void unregister(Subject subject,
82  IObservable observer) {
83      map.get(subject).remove(observer);
84  }
85
86  /**
87   * 根据特定主题通知监听器
88   *
89   * @param subject
90   */
91  public synchronized void notifyBySubject(Subject subject,
92  Object args) {
93      for (IObservable observer : map.get(subject)) {
94          observer.update(subject, args);
95      }
96  }
97
98  }

```

代码片段17是这个体系中最复杂的一个类，且这个类是单例类，此类的主要功能是：

- 管理主题对象和其观察者的映射关系。
- 根据主题对象的要求进行观察者的通知行为。
- 维持多线程情况下的安全性。

客户端代码：

代码片段18 Client.java

```

1  package cn.steven.pattern.demo.observer;
2
3  /**
4   * 更改管理器客户端
5   */
6  public class Client {
7
8      public static void main(String[] args) {
9
10         /**
11          * 初始化主题对象
12          */
13         NewsPublisher newsPublisher = new NewsPublisher();
14         GoodsPublisher goodsPublisher = new GoodsPublisher();
15
16         /**
17          * 初始化监听器
18          */
19         IObservable observerA = new Customer();

```



```
20         IObservable observerB = new TV();
21
22         /**
23          * 注册监听器，注意两个注册的不同组合
24          */
25         newsPublisher.addObserver(observerA);
26
27         goodsPublisher.addObserver(observerA);
28         goodsPublisher.addObserver(observerB);
29
30         /**
31          * 触发改变
32          */
33         newsPublisher.setNews("新闻：新店开张！");
34         goodsPublisher.setNews("商品：新货上架！");
35
36     }
37
38 }
```

运行结果如图17-11所示。

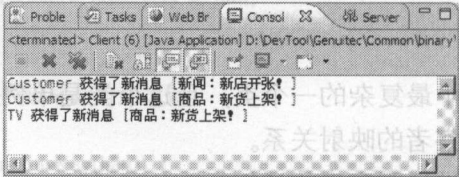


图17-11 运行结果

由图17-11可见，通过更改管理器，多个主题与多个观察者之间就可以很好地进行联系了，且外部代码也可以随时观察所有的映射状态。

17.2.3 观察者模式在JDK中的实例

JDK中提供了很多观察者模式的具体辅助类和实例，如java.util.Observable抽象类和java.util.Observer接口，它们组成了基本观察者模式的基础。

Observable抽象类的功能说明如下：

public class Observable extends Object

此类表示模型视图范例中的Observable对象，或者说“数据”。可将其子类化，以表示应用程序想要观察的对象。

一个Observable对象可以有一个或多个观察者。观察者可以是实现了Observer接口的任意对象。一个Observable实例改变后，调用Observable的notifyObservers方法的应用程序会通过调用观察者的update方法来通知观察者该实例发生了改变。

Observable类中所提供的默认实现将按照其注册的重要性顺序来通知Observers，但是子类可能改变此顺序，从而使用非固定顺序在单独的线程上发送通知，或者也可以保证其子类遵从其所选择的顺序。

注意，此通知机制与线程无关，并且与Object类的wait和notify机制完全独立。

新创建一个Observable对象时，其观察者集是空的。当且仅当equals方法为两个观察者返回true时，才认为它们是相同的。

从以下版本开始：JDK 1.0

Observer接口说明如下：

public interface Observer

一个可在观察者要得到Observable对象更改通知时可实现Observer接口的类。

从以下版本开始：JDK 1.0

另请参见：

Observable

使用情况举例如下：

代码片段19 JDKObserverExample.java

```
1 package cn.steven.pattern.demo.observer.jdk;
2
3 import java.util.Observable;
4 import java.util.Observer;
5
6 /**
7  * 使用java.util.Observable及java.util.Observer实现观察者模式的案例
8  */
9 public class JDKObserverExample {
10     public static void main(String[] args) {
11         /**
12          * 建立主题对象
13          */
14         MySubject subject = new MySubject();
15
16         /**
17          * 监听关系
18          */
19         subject.addObserver(new MyObserver());
20
21         /**
22          * 通知
23          */
24         subject.setContent("下雨收衣服了~");
25
26         //运行结果:
27         //通知来自cn.steven.pattern.demo.observer.jdk.MySubject:下雨收衣服了~
28     }
29 }
30
31 /**
32  * 主题类
33  */
34 }
```

```
35 class MySubject extends Observable {
36
37     /**
38      * 通知内容
39      */
40     private String content;
41
42     public void setContent(String content) {
43         this.content = content;
44
45         /**
46          * 表示内容已经更改需要通知
47          */
48         this.setChanged();
49
50         /**
51          * 调用通知方法
52          */
53         this.notifyObservers(content);
54     }
55
56 }
57
58 /**
59  * 监听器类
60  */
61 class MyObserver implements Observer {
62
63     @Override
64     public void update(Observable o, Object arg) {
65         System.out.println("通知来自"
66             + o.getClass().getName() + ":" + arg);
67     }
68
69 }
```

由代码片段19可见使用JDK提供的类实现观察者模式是十分方便的。通常情况下，建议使用这种方式实现。

有时候，如果通知消息比较复杂的话，使用代码片段19会感觉如果让JDK代码管理映射会不灵活，这时可以自己管理消息的映射，JDK提供了一种简单的机制来完成这种观察者模式。使用的是java.util.EventListener——事件监听/处理接口和java.util.EventObject——事件（状态）对象根类，此处事件监听/处理接口就起到了监听器的作用，而事件则起到了一种比主题更细粒度的控制的作用，比如一个主题可以有多种消息需要更新，而每一种消息都需要使用特定的监听器的话，这种基于事件/监听的模式就比较适合了，在java.awt和javaw.swing包中就是这样设计的。

事件/监听的文档描述：

注意：此通知机制与线程无关，并且与Observable的notify机制完全独立。

public class EventObject extends Object implements Serializable

所有事件状态对象都将从其根类派生。

所有Event在构造时都引用了对象“source”，在逻辑上我们认为该对象是最初发生有关事件的对象。

从以下版本开始：JDK 1.1。

另请参见：序列化表格。

public interface EventListener

所有事件监听器接口必须扩展的标记接口。

从以下版本开始：JDK 1.1。

使用情况举例如下：

代码片段20 JDKEventExample.java

```

1  package cn.steven.pattern.demo.observer.jdk;
2
3  import java.util.Date;
4  import java.util.EventListener;
5  import java.util.EventObject;
6  import java.util.Vector;
7
8  /**
9   * 基于事件/监听器的例子
10  */
11  public class JDKEventExample {
12
13      public static void main(String[] args) {
14          /**
15           * 信息主题
16           */
17          MyObject object = new MyObject();
18
19          /**
20           * 增加监听器
21           */
22          object.addListener(new MyListener());
23          // 匿名类监听器
24          object.addListener(new IMyEventListener() {
25
26              @Override
27              public void doListener(MyEvent myEvent) {
28                  System.out.println("匿名类: " + myEvent.getMessage());
29              }
30          });
31
32          /**
33           * 触发事件
34           */
35          object.doEvent();
36      }
37  }

```

```

36 }
37
38 /**
39  * 监听器接口
40  */
41 interface IMyEventListener extends EventListener {
42     void doListener(MyEvent myEvent);
43 }
44
45 /**
46  * 监听器
47  */
48 class MyListener implements IMyEventListener {
49
50     @Override
51     public void doListener(MyEvent myEvent) {
52         System.out.println("MyListener接收到新事件: "
53             + myEvent.getMessage());
54     }
55 }
56
57 /**
58  * 自定义事件
59  */
60 class MyEvent extends EventObject {
61
62     public String getMessage() {
63         return "新的时间通知: " + new Date().toLocaleString();
64     }
65
66     /**
67      * 构造方法
68      *
69      * @param source
70      *      产生事件的源对象
71      */
72     public MyEvent(Object source) {
73         super(source);
74     }
75
76 }
77
78 class MyObject {
79     /**
80      * 如果需要监听多种事件需要使用Map
81      */
82     private Vector<IMyEventListener> el
83         = new Vector<IMyEventListener>();
84
85     /**
86      * 增加监听器

```

```
87 */
88 public void addListener(IMyEventListener myEventListener) {
89     if (!el.contains(myEventListener)) {
90         el.add(myEventListener);
91     }
92 }
93
94 /**
95  * 删除监听器
96  */
97 public void removeListener(IMyEventListener myEventListener) {
98     el.remove(myEventListener);
99 }
100
101 /**
102  * 通知事件
103  */
104 public void notifyEvent(MyEvent myEvent) {
105     for (IMyEventListener myEventListener : el) {
106         myEventListener.doListener(myEvent);
107     }
108 }
109
110 /**
111  * 触发事件
112  */
113 public void doEvent() {
114     notifyEvent(new MyEvent(this));
115 }
116 }
```

运行结果如图17-12所示。

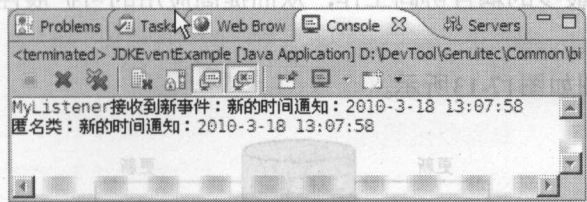


图17-12 运行结果

由图17-12可见，基于事件/监听器的模型与基于主题/观察者的模型的功能十分相似，在实际应用中，复杂的情况下通常用基于事件的模型，这也是大部分图形化客户端实现的机制。在架构基于多种视图的应用时经常会用到观察者模式。

17.2.4 观察者模式的使用范围

观察者模式的应用场景：

- 对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。
- 对象仅需要将将自己的更新通知给其他对象而不需要知道其他对象的细节。

该模式的典型应用：

- 监听事件驱动程序设计中的外部事件。
- 监听/监视某个对象的状态变化。
- 发布者/订阅者（publisher/subscriber）模型中，当一个外部事件（新的产品，消息的出现等）被触发时，通知邮件列表中的订阅者。

该模式的优点：

- 对象之间可以进行同步通信。
- 可以同时通知一到多个关联对象。
- 对象之间的关系以松耦合的形式组合，互不依赖。

观察者模式的缺陷：

- 松耦合导致代码关系不明显，有时可能难以理解。
- 如果一个被观察者对象有很多直接或间接的观察者的话，将所有的观察者都通知到会花费很多时间。
- 如果在被观察者之间有循环依赖的话，被观察者会触发循环调用，容易导致系统崩溃，在使用此模式时应特别注意这一点。
- 如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以不会引起内部冲突的方式进行的。
- 虽然观察者模式可以使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制让观察者知道所观察的对象是怎么发生变化的。

17.2.5 与其他模式的关系

观察者模式中可以使用单例模式来构建一个更改管理器对象。

MVC模式是所有面向对象程序设计语言都应该遵守的规范。

MVC模式将一个应用分成三个基本部分：Model（模型）、View（视图）和Controller（控制器），这三个部分以最少的耦合协同工作，从而提高应用的可扩展性及可维护性。此模式可以用观察者模式来实现。

MVC模式的架构图如图17-13所示。

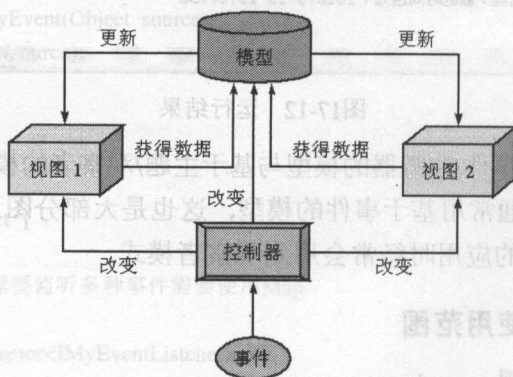


图17-13 MVC模式

在经典的MVC模式中，事件由控制器处理，控制器根据事件的类型改变模型或视图，反之亦然。具体地说，每个模型对应一系列的视图列表，这种对应关系通常采用注册来完成，即将多个视图注册到同一个模型，当模型发生改变时，模型向所有注册过的视图发送通知，接下来，视图从对应的模型中获得信息，然后完成视图显示的更新。

从设计模式的角度来看，MVC思想非常类似于一个观察者模式，但与观察者模式存在少许差别：观察者模式下，观察者和被观察者可以是两个互相对等的对象，但对于MVC思想而言，被观察者往往只是单纯的数据体，而观察者则是单纯的视图页面。

17.3 观察者模式总结

在面向对象的设计过程中，依赖关系必不可少，我们经常在为实现对象间的松耦合设计而努力，这是我们进行OO设计的又一个准则，除了面向接口编程以外，还需要做的是把是一个变为有一个 (is a to has a) 。

观察者模式的适用范围，就如定义所言，适用于一对多的依赖关系，其一个对象状态的改变将影响所有其他对象，比如在银行系统中，客户账户的利息的计算是依赖于银行的利率的，银行的利率的改变，必须由银行通知所有的客户，然后根据新利息进行计算，结果也会发生变化。这里，银行相当于一个主题，银行客户就是观察者，他们会订阅这个主题。在这里可以应用观察者模式。

读者应该善于在需求中发现这种依赖并学会将一种合适的观察者模式的实现方法应用进去。

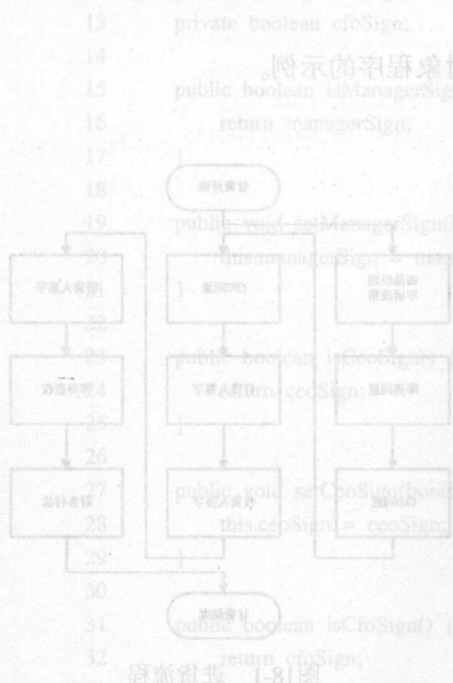


图 17-1 观察者模式

第18章 责任链模式 (Chain of Responsibility)

责任链模式使多个对象都有机会处理请求 (Request)，从而避免请求的发送者和接收者之间的耦合关系。该模式将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理请求为止¹。

在生活中，经常会发生某种职责处理的问题。比如在一家公司里，某位员工小王的工资少了一千元，他非常不解，于是将这件事反映给了老板，老板一定要给小王一个交代，于是找来了财务总监询问此事，财务总监不清楚此事于是询问了工资核算的员工，工资核算的员工查过了工资记录，发现是人事部门本月给小王扣了一千元的工资，于是询问了人事总监，人事总监不清楚这件事，于是问了人事处的稽查科，稽查科的科长知道这件事是因为小王本月旷工了三天，所以才有此决定。

经过上面的例子可以看出，如果有众多对象存在，则定位负责某事的对象是一件困难的事。面向对象开发人员通常希望明确和减少对象间的责任，从而降低对象之间的耦合度。这样我们的系统更加容易修改，同时也可降低产生缺陷的风险。从某种程度上说，Java语言本身就能够帮助降低对象间的耦合度。客户端只能访问对象的接口，而不用关心其具体实现细节。借助于接口这种方式，客户端只需要了解方法的功能即可。如果我们按照某种层次结构进行组织，比如组织为责任链，客户端代码就有可能不用事先了解自己将使用哪一个类。在这种情况下，链中的每个对象都有个方法，当客户端代码调用该方法的时候，这些对象要么执行该方法，要么沿着这条链转发该方法调用请求。

以上便是使用生活中的责任链模式来设计面向对象程序的示例。

18.1 商品进货审批中的问题

连锁店的进货审批流程是十分复杂的，必须要经过几道手续后才能审批完成。参与审批的人数比较多，而且每个人都需要按照顺序签字，流程如图18-1所示。

由图18-1可见，进货的流程是十分复杂的，而且这样的流程在不同的环境、不同的公司可能都是不一样的，这样就极大地增加了编程实现的难度。

以下的一个初步编程实现中为了简化流程只使用了商品经理、CEO、CFO这三个角色审批，最简化的一个实现是依次让这三个人

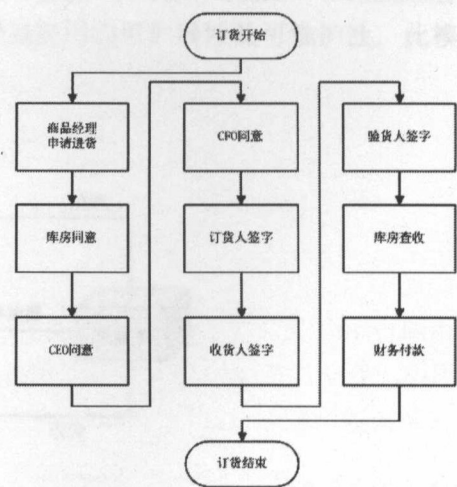


图18-1 进货流程

¹Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.GOF[95]

进行审批即可，序列图如图18-2所示。

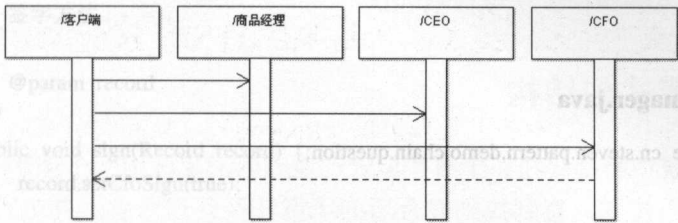


图18-2 进货流程序列图

实现代码如下，需要审批的单据：

代码片段1 Record.java

```
1 package cn.steven.pattern.demo.chain.question;
2
3 /**
4  * 需要审批的单据
5  */
6 public class Record {
7
8     /**
9      * 表示各个职位的人是否已经签字
10     */
11     private boolean managerSign;
12     private boolean ceoSign;
13     private boolean cfoSign;
14
15     public boolean isManagerSign() {
16         return managerSign;
17     }
18
19     public void setManagerSign(boolean managerSign) {
20         this.managerSign = managerSign;
21     }
22
23     public boolean isCeoSign() {
24         return ceoSign;
25     }
26
27     public void setCeoSign(boolean ceoSign) {
28         this.ceoSign = ceoSign;
29     }
30
31     public boolean isCfoSign() {
32         return cfoSign;
33     }
34
35     public void setCfoSign(boolean cfoSign) {
36         this.cfoSign = cfoSign;
37     }
38 }
```

38
39 }

商品经理:

代码片段2 Manager.java

```
1 package cn.steven.pattern.demo.chain.question;  
2  
3 /**  
4  * 经理  
5  */  
6 public class Manager {  
7  
8     /**  
9     * 签字方法  
10    *  
11    * @param record  
12    */  
13    public void sign(Record record) {  
14        record.setManagerSign(true);  
15    }  
16 }
```

CEO:

代码片段3 CEO.java

```
1 package cn.steven.pattern.demo.chain.question;  
2  
3 /**  
4  * CEO  
5  */  
6 public class CEO {  
7  
8     /**  
9     * 签字方法  
10    *  
11    * @param record  
12    */  
13    public void sign(Record record) {  
14        record.setCeoSign(true);  
15    }  
16 }
```

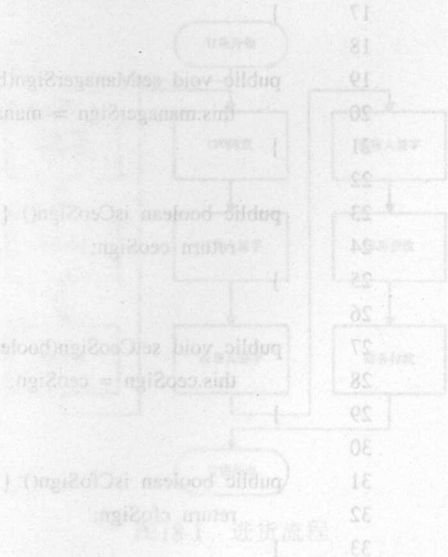
CFO:

代码片段4 CFO.java

```
1 package cn.steven.pattern.demo.chain.question;  
2  
3 /**  
4  * CFO  
5  */  
6 public class CFO {
```

责任链模式 (Chain of Responsibility)

图8-18 责任链模式



```
7
8 /**
9  * 签字方法
10  *
11  * @param record
12  */
13 public void sign(Record record) {
14     record.setCfoSign(true);
15 }
16 }
```

客户代码:

代码片段5 Client.java

```
1 package cn.steven.pattern.demo.chain.question;
2
3
4 /**
5  * 客户代码
6  */
7 public class Client {
8
9     public static void main(String[] args) {
10
11         /**
12          * 实例化需要的对象
13          */
14         Record record = new Record();
15         Manager manager = new Manager();
16         CEO ceo = new CEO();
17         CFO cfo = new CFO();
18
19         /**
20          * 按流程处理
21          */
22         manager.sign(record);
23         ceo.sign(record);
24         cfo.sign(record);
25
26         /**
27          * 结束
28          */
29     }
30 }
31
32 }
```

以上代码的流程十分清晰,已经可以满足简单情况下的需求,但是如果使用情况比较复杂后,就会发现有如下问题:

- 多个审阅者无法自行调用自身应该调用的方法。
- 客户操作需显式指定调用顺序及调用的方法,使得代码非常复杂。

• 审阅者的困难增加，无法动态生成一组对象来操作审阅过程。
以上的问题就可以通过责任链模式的设计方法进行解决。

18.2 责任链模式的结构

责任链模式系统中将会存在多个有相似处理能力的对象。当一个请求触发后，请求将在这些对象组成的链条中传递，直到找到最合适的“责任”处理对象，并进行处理。

18.2.1 责任链模式

责任链模式的主要实现思路是将所有处理请求的对象组成一个链条，纯粹的责任链模式的每一个处理对象能做两件事情：处理请求并返回或将请求传递给链条上的下一个对象。

为了演示此模式的功能，下面的例子以一个算术题目的需求作为演示。由用户给出一个算术题目（数字1、数字2、操作符号），然后传递给一个进行处理的链条，最终得到结果。
其设计类图如图18-3所示。

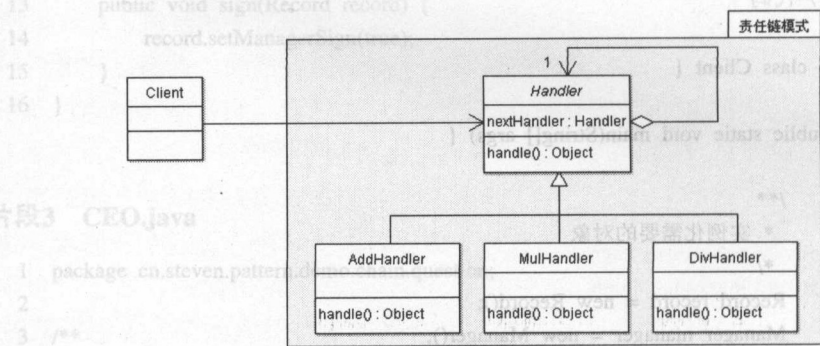


图18-3 责任链模式类图

图中的参与者如下：

- 抽象处理者（Handler）角色：定义出一个处理请求的接口，维护一个自身的对象，如果需要，接口可以定义出一个方法，以返回对下家的引用。
- 具体处理者（xxxHandler）角色：接到请求后，可以选择将请求处理掉，或者调用父类方法将请求传给下家。
- 客户端（Client）：构造责任链模式的处理链条，将命令传递给链条。

注意在所有处理类的抽象类中由于引入了一个自身聚合的nextHandler的对象，所以一个具体的处理的对象可以维护着链条上的下一个对象，这种结构类似于单链表¹。

实现代码如下，处理抽象类：

代码片段6 Handler.java

```
1 package cn.steven.pattern.demo.chain.pattern;
2
3 /**
```

¹<http://zh.wikipedia.org/zh-cn/%E5%8D%95%E9%93%BE%E8%A1%A8>.

```
4  * 责任链模式抽象类
5  */
6  public abstract class Handler {
7
8      /**
9       * 存放链条的下一个节点对象
10     */
11     private Handler nextHandler;
12
13     public Handler getNextHandler() {
14         return nextHandler;
15     }
16
17     public void setNextHandler(Handler nextHandler) {
18         this.nextHandler = nextHandler;
19     }
20
21     /**
22     * 默认的处理需求的方法
23     */
24     public Object handle(double num1, double num2,
25         String calculateType) {
26         if (nextHandler == null) {
27             return null;
28         } else {
29             return nextHandler.handle(num1, num2, calculateType);
30         }
31     }
32 }
33 }
```

实现加法的处理类:

代码片段7 AddHandler.java

```
1  package cn.steven.pattern.demo.chain.pattern;
2
3  /**
4   * 具体处理类: 加法
5   */
6  public class AddHandler extends Handler {
7
8      /**
9       * 重写父类的方法
10     */
11     @Override
12     public Object handle(double num1, double num2,
13         String calculateType) {
14         /**
15          * 如果是加法就处理, 如果不是就调用父类的默认处理方法
16          */
17         if (calculateType.equals("+")) {
```

```
18         return num1 + num2;
19     } else {
20         return super.handle(num1, num2, calculateType);
21     }
22 }
23 }
```

实现乘法的处理类:

代码片段8 MulHandler.java

```
1 package cn.steven.pattern.demo.chain.pattern;
2
3 /**
4  * 具体处理类: 乘法
5  */
6 public class MulHandler extends Handler {
7
8     /**
9      * 重写父类的方法
10     */
11     @Override
12     public Object handle(double num1, double num2,
13         String calculateType) {
14         /**
15          * 如果是乘法就处理, 如果不是就调用父类的默认处理方法
16          */
17         if (calculateType.equals("*")) {
18             return num1 * num2;
19         } else {
20             return super.handle(num1, num2, calculateType);
21         }
22     }
23 }
```

实现除法的处理类:

代码片段9 DivHandler.java

```
1 package cn.steven.pattern.demo.chain.pattern;
2
3 /**
4  * 具体处理类: 除法
5  */
6 public class DivHandler extends Handler {
7
8     /**
9      * 重写父类的方法
10     */
11     @Override
12     public Object handle(double num1, double num2,
13         String calculateType) {
14         /**
```



```
15  * 如果是除法就处理，如果不是就调用父类的默认处理方法
16  */
17  if (calculateType.equals("/")) {
18      return num1 / num2;
19  } else {
20      return super.handle(num1, num2, calculateType);
21  }
22  }
23 }
```

由以上三个具体的实现类可见，每一个类只处理自己可以处理的一类操作，如果不能操作则会调用父类的处理方法，也就是交给链条的下一个节点处理。

客户端代码：

代码片段10 Client.java

```
1 package cn.steven.pattern.demo.chain.pattern;
2
3 /**
4  * 责任链模式客户端
5  */
6 public class Client {
7     public static void main(String[] args) {
8         /**
9          * 创建处理对象
10          */
11         Handler tophandler = new AddHandler();
12         Handler mulhandler = new MulHandler();
13         Handler divhandler = new DivHandler();
14
15         /**
16          * 组装链条，结构如下
17          * +-----+ +-----+ +-----+
18          * | tophandler +----> mulhandler +----> divhandler |
19          * +-----+ +-----+ +-----+
20          */
21         mulhandler.setNextHandler(divhandler);
22         tophandler.setNextHandler(mulhandler);
23
24         /**
25          * 处理需求
26          */
27         System.out.println("34*23=" + tophandler.handle(34, 23, "*"));
28         System.out.println("13/3=" + tophandler.handle(34, 23, "/"));
29     }
30
31 }
```

由客户端代码可以看出责任链的构造过程。运行时的序列如图18-4所示。注意应将图18-4和图18-2进行对比理解。运行结果如图18-5所示。

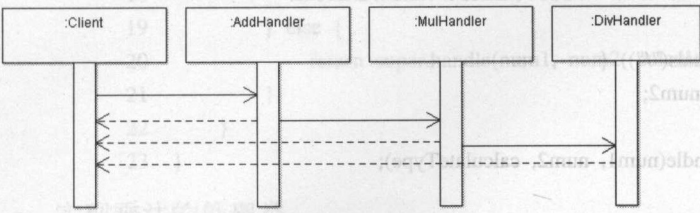


图18-4 责任链模式序列图

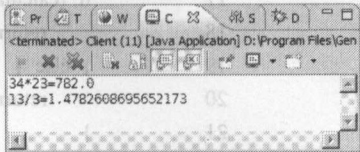


图18-5 运行结果

由运行结果可见，只要是责任链条上支持的操作，都可以被正确执行。客户无需关心具体是链条上的哪一个对象来执行处理。

18.2.2 纯的和纯的责任链模式

上一节介绍的责任链模式就是一种纯的责任链模式。一个纯的责任链模式要求一个具体的处理者对象只能在两个行为中选择一个：一是承担责任，二是把责任推给下家。而不允许出现某一个具体处理者对象在承担了一部分责任后又把责任向下传的情况。

在一个纯的责任链模式里面，一个请求必须被某一个处理者对象所接受；在一个不纯的责任链模式里面，一个请求能最终不被所有接收端对象所接受。

纯的责任链模式的实际例子非常难找到，一般看到的例子均是不纯的责任链模式的实现。有些人认为不纯的责任链根本不是责任链模式，这也许是有道理的。不过在实际的系统里，纯的责任链非常难找到。如果坚持责任链不纯便不是责任链模式，那么责任链模式便不会有太大的意义了。

18.2.3 用责任链模式建立明晰的进货审批流程

根据责任链模式的设计方式，下面就可以解决进货审批的流程问题了，由于在此过程中有很多对象都要操作审批的单据，所以使用的是一种不纯的责任链模式。

可以发现类似于代码片段10的责任链创建方式不能很灵活地应对经常发生变动的需要，如果要更改责任链的构成对象时，必须要修改源代码才行，所以在下面的例子中使用了XML¹配置责任链的方式，这样一旦需要修改的话，只需要修改XML文件即可。

XML的解析方法使用的是JAXP²中的DOM解析方式。

最终设计的类图如图18-6所示。

代码展示如下：

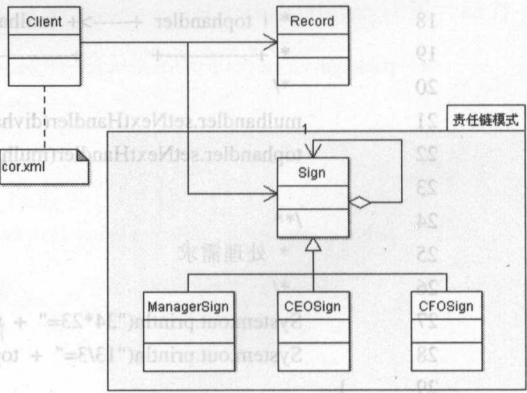


图18-6 设计类图

¹<http://zh.wikipedia.org/zh-cn/Xml>.
²<http://www.ibm.com/developerworks/java/library/x-jaxp/>.

表示审批单据的类:

代码片段11 Record.java

```
1 package cn.steven.pattern.demo.chain.question;
2
3 /**
4  * 需要审批的单据
5  */
6 public class Record {
7
8     /**
9      * 表示各个职位的人是否已经签字
10     */
11     private boolean managerSign;
12     private boolean ceoSign;
13     private boolean cfoSign;
14
15     public boolean isManagerSign() {
16         return managerSign;
17     }
18
19     public void setManagerSign(boolean managerSign) {
20         this.managerSign = managerSign;
21     }
22
23     public boolean isCeoSign() {
24         return ceoSign;
25     }
26
27     public void setCeoSign(boolean ceoSign) {
28         this.ceoSign = ceoSign;
29     }
30
31     public boolean isCfoSign() {
32         return cfoSign;
33     }
34
35     public void setCfoSign(boolean cfoSign) {
36         this.cfoSign = cfoSign;
37     }
38
39 }
```

签字的抽象类:

代码片段12 Sign.java

```
1 package cn.steven.pattern.demo.chain;
2
3 import cn.steven.pattern.demo.chain.question.Record;
4
5 /**
```



```
6 * 审批抽象类
7 */
8 public abstract class Sign {
9
10     /**
11      * 下一个签字者
12      */
13     private Sign next;
14
15     public Sign getNext() {
16         return next;
17     }
18
19     public void setNext(Sign next) {
20         this.next = next;
21     }
22
23     /**
24      * 签字方法
25      * @param record
26      */
27     public void sign(Record record) {
28         if (next == null) {
29             return;
30         } else {
31             next.sign(record);
32         }
33     }
34 }
```

签字类（经理）：

代码片段13 ManagerSign.java

```
1 package cn.steven.pattern.demo.chain;
2
3 import cn.steven.pattern.demo.chain.question.Record;
4
5 /**
6  * 签字者：经理
7  */
8 public class ManagerSign extends Sign {
9
10     /**
11      * 重写签字方法
12      */
13     @Override
14     public void sign(Record record) {
15         /**
16          * 在指定的位置签字
17          */
18         record.setManagerSign(true);
```

```
19
20 /**
21  * 调用父类的签字方法进行下一个人员的签字
22  */
23 super.sign(record);
24 }
25 }
```

签字类 (CEO) :

代码片段14 CEOSign.java

```
1 package cn.steven.pattern.demo.chain;
2
3 import cn.steven.pattern.demo.chain.question.Record;
4
5 /**
6  * 签字者: CEO
7  */
8 public class CEOSign extends Sign {
9
10    /**
11     * 重写签字方法
12     */
13    @Override
14    public void sign(Record record) {
15        /**
16         * 在指定的位置签字
17         */
18        record.setCeoSign(true);
19
20        /**
21         * 调用父类的签字方法进行下一个人员的签字
22         */
23        super.sign(record);
24    }
25 }
```

签字类 (CFO) :

代码片段15 CFOSign.java

```
1 package cn.steven.pattern.demo.chain;
2
3 import cn.steven.pattern.demo.chain.question.Record;
4
5 /**
6  * 签字者: CFO
7  */
8 public class CFOSign extends Sign {
9
10    /**
11     * 重写签字方法
12     */
```

```

13     @Override
14     public void sign(Record record) {
15         /**
16          * 在指定的位置签字
17          */
18         record.setCfoSign(true);
19
20         /**
21          * 调用父类的签字方法进行下一个人员的签字
22          */
23         super.sign(record);
24     }
25 }

```

下面展示的是XML文件配置，此配置展示了构建责任链类的信息：

代码片段16 cor.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!--此文档定义了责任链模式中构成链的类名-->
3 <chain>
4     <handler class="cn.steven.pattern.demo.chain.ManagerSign"/>
5     <handler class="cn.steven.pattern.demo.chain.CEOSign"/>
6     <handler class="cn.steven.pattern.demo.chain.CFOSign"/>
7 </chain>

```

使用JXAP的Client类：

代码片段17 Client.java

```

1 package cn.steven.pattern.demo.chain;
2
3 import javax.xml.parsers.DocumentBuilder;
4 import javax.xml.parsers.DocumentBuilderFactory;
5
6 import org.w3c.dom.Document;
7 import org.w3c.dom.NamedNodeMap;
8 import org.w3c.dom.Node;
9 import org.w3c.dom.NodeList;
10
11 import cn.steven.pattern.demo.chain.question.Record;
12
13 /**
14  * XML配置责任链模式客户端
15  */
16 public class Client {
17
18     public static void main(String[] args) throws Exception {
19
20         /**
21          * 解析XML文件
22          */
23         DocumentBuilder documentBuilder = DocumentBuilderFactory

```



```
24         .newInstance().newDocumentBuilder();
25
26         // System.out.println(documentBuilder);
27
28         Document document = documentBuilder
29             .parse(Client.class
30                 .getResourceAsStream(
31                     "/cn/steven/pattern/demo/chain/cor.xml"));
32
33         NodeList nodeList = document.getElementsByTagName("handler");
34
35         /**
36          * 创建链条顶点
37          */
38         Sign top = null;
39         /**
40          * 链条末端
41          */
42         Sign last = null;
43
44         /**
45          * 获取链中的类进行装配
46          */
47         for (int i = 0; i < nodeList.getLength(); i++) {
48             Node item = nodeList.item(i);
49             NamedNodeMap attributes = item.getAttributes();
50             Node attr = attributes.getNamedItem("class");
51             // System.out.println(attr.getNodeValue());
52
53             /**
54              * 装配
55              */
56             if (top == null) {
57                 /**
58                  * 顶点
59                  */
60                 top = last = (Sign) Class
61                     ..forName(attr.getNodeValue()).newInstance();
62             } else {
63                 /**
64                  * 其他点
65                  */
66                 Sign chain = (Sign) Class
67                     .forName(attr.getNodeValue()).newInstance();
68                 last.setNext(chain);
69                 last = chain;
70             }
71         }
72     }
73
74     /**
```

```

75      * 操作签字
76      */
77      Record record = new Record();
78      top.sign(record);
79
80      System.out.println("manager签字: " + record.isManagerSign());
81      System.out.println("CEO签字: " + record.isCeoSign());
82      System.out.println("CFO签字: " + record.isCfoSign());
83
84  }
85
86  }

```

运行结果如图18-7所示。

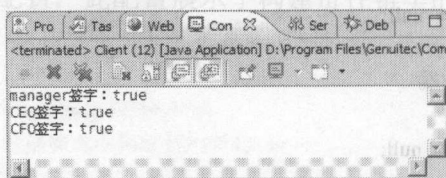


图18-7 运行结果

如图18-7可见现在客户端就可以通过简单的方式执行步骤烦琐的审批任务了，而且由于XML配置文件的引入，更改责任链的结构更加容易，只需要使用文本编辑器改写cor.xml文件即可。

18.2.4 责任链模式在JDK中的实例

Java 1.0版中，Awt库使用了责任链模式和命令模式来处理用户界面的事件。由于用户界面的组成通常是由一层一层的容器控件包含具体控件的形式组成的，因此当事件发生在一个部件上时，此部件的事件处理器可以处理此事件，然后决定是否将事件向上级容器部件传播；上级容器部件接到事件后可以处理此事件，然后决定是否将事件再次向上级容器部件传播，如此往复，直到事件到达顶层部件。目前JavaScript也是采用此种方法处理事件的¹。

这种处理事件的方式有如下缺点：

- Awt 1.0的事件处理的模型是基于继承的。为了使一个程序能够捕捉GUI的事件并处理此事件，必须subclass此部件并且给其子类配备事件处理器，也就是置换掉action()方法或者handleEvent()方法。这不是应当提倡的做法：在一个面向对象的系统里，经常使用的应当是委派，继承不应当是常态。在一个复杂的GUI系统里，像这样为所有有事件的部件提供子类，会导致很多的子类，这不是很麻烦吗？当然，基于事件浮升机制，可以在部件的树结构的根部部件里面处理所有的事件。但是这样一来，就需要使用复杂的条件转移语句在这个根部部件里辨别事件的起源和处理方法。这种非常过程化的处理方法很难维护，并且与面向对象的设计思想相违背。

- 由于每一个事件都会沿着部件树结构向上传播，因此事件浮升机制会使得事件的处理变得较慢。这也是缺点之一。比如在有些操作系统中，鼠标每移动一个色素，都会激发一个MOUSE_MOVE事件。每一个这样的事件都会沿着部件的容器树结构向上传播，这会使得鼠标

¹http://www.itlearner.com/code/js_ref/evnt.htm.

事件成灾。

在Java中，Awt 1.0的事件处理的模型只适用于Awt部件类。这是此模型的另一个缺点。责任链模式要求链上所有的对象都继承自一个共同的父类，这个类便是java.awt.Component类。

显然，由于每一级的部件在接到事件时，都可以处理此事件，而不论此事件是否在这一级得到处理，事件都可以停止向上传播或者继续向上传播。这是典型的不纯的责任链模式。

自Awt 1.1版以后，Awt的事件处理模型与1.0相比有了很大的变化。新的事件处理模型是建立在观察者模式的基础之上的，而不再是责任链模式的基础之上。

目前的事件传播机制是基于监听器的观察者模式，其实例代码如下：

代码片段18 SwingTest.java

```
1 package cn.steven.pattern.demo.chain.jdk;
2
3 import java.awt.Color;
4 import java.awt.FlowLayout;
5 import java.awt.HeadlessException;
6 import java.awt.event.MouseAdapter;
7 import java.awt.event.MouseEvent;
8
9 import javax.swing.JButton;
10 import javax.swing.JFrame;
11 import javax.swing.JOptionPane;
12 import javax.swing.JPanel;
13 import javax.swing.SwingUtilities;
14
15 /**
16  * 观察者模式实现的事件机制
17  */
18 public class SwingTest {
19     public static void main(String[] args) {
20         SwingUtilities.invokeLater(new Runnable() {
21             @Override
22             public void run() {
23                 new MyWindow();
24             }
25         });
26     }
27 }
28
29 class MyWindow extends JFrame {
30     private JPanel jpa;
31     private JPanel jpb;
32     private JButton jba;
33
34     public MyWindow() throws HeadlessException {
35         super();
36         this.setTitle("测试窗体");
37         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
38         // this.setSize(400, 300);
39     }
40 }
```



```

40         new ArrayList<Address>();
41         add();
42         merge();
43         getAddress();
44     }
45
46     public void merge() {
47         new ArrayList<Address>();
48         add();
49         merge();
50         getAddress();
51     }
52
53     public void merge() {
54         new ArrayList<Address>();
55         add();
56         merge();
57         getAddress();
58     }
59
60     public void merge() {
61         new ArrayList<Address>();
62         add();
63         merge();
64         getAddress();
65     }
66
67     public void merge() {
68         new ArrayList<Address>();
69         add();
70         merge();
71         getAddress();
72     }
73
74     public void merge() {
75         new ArrayList<Address>();
76         add();
77         merge();
78         getAddress();
79     }
80
81     public void merge() {
82         new ArrayList<Address>();
83         add();
84         merge();
85         getAddress();
86     }
87
88     public void merge() {
89         new ArrayList<Address>();
90         add();
91         merge();
92         getAddress();
93     }
94
95     public void merge() {
96         new ArrayList<Address>();
97         add();
98         merge();
99         getAddress();
100    }

```

图 18-8 图 18-7 的另一种实现

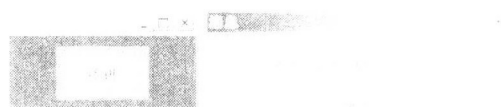


图 18-8

同样的代码如果使用Awt 1.0机制实现的话将会依次触发控件及其各个容器的单击事件。在JavaEE中，也有使用责任链模式的例子，请观察下面的HttpServlet的代码片段：

代码片段19 HttpServlet.java片段

```
1 protected void service(HttpServletRequest req,
2                          HttpServletResponse resp)
3     throws ServletException, IOException {
4
5     String method = req.getMethod();
6
7     if (method.equals(METHOD_GET)) {
8         long lastModified = getLastModified(req);
9         if (lastModified == -1) {
10             // servlet doesn't support if-modified-since, no reason
11             // to go through further expensive logic
12             doGet(req, resp);
13         } else {
14             long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
15             if (ifModifiedSince < (lastModified / 1000 * 1000)) {
16                 // If the servlet mod time is later, call doGet()
17                 // Round down to the nearest second for a proper compare
18                 // A ifModifiedSince of -1 will always be less
19                 maybeSetLastModified(resp, lastModified);
20                 doGet(req, resp);
21             } else {
22                 resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
23             }
24         }
25     } else if (method.equals(METHOD_HEAD)) {
26         long lastModified = getLastModified(req);
27         maybeSetLastModified(resp, lastModified);
28         doHead(req, resp);
29     } else if (method.equals(METHOD_POST)) {
30         doPost(req, resp);
31     } else if (method.equals(METHOD_PUT)) {
32         doPut(req, resp);
33     } else if (method.equals(METHOD_DELETE)) {
34         doDelete(req, resp);
35     } else if (method.equals(METHOD_OPTIONS)) {
36         doOptions(req, resp);
37     } else if (method.equals(METHOD_TRACE)) {
38         doTrace(req, resp);
39     } else {
40         //
```

```

48 // Note that this means NO servlet supports whatever
49 // method was requested, anywhere on this server.
50 //
51
52 String errMsg = lStrings.getString(
53     "http.method_not_implemented");
54 Object[] errArgs = new Object[1];
55 errArgs[0] = method;
56 errMsg = MessageFormat.format(errMsg, errArgs);
57
58 resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
59 }
60 }

```

由代码片段19片段可见，此类的service方法即为责任链模式接口中的处理方法，只不过此类中对每一种do方法都给出了默认实现，比如doPost、doGet、doDelete等，这样就方便了其子类的编程过程。

子类只需要针对某种特定的do方法编程即可，此种责任链的生成方式适用于要处理的方法个数固定的情形。

18.2.5 责任链模式的使用范围

责任链模式特征：

- 处理请求的对象集合（set of potential request handler objects）以及它们在链表中的顺序是由客户端根据现在应用的状态在运行时动态决定的。

- 客户端根据现在的状态，对于不同的请求类型，可以拥有不同的可能处理请求的对象集合（set of potential request handler objects）。一个处理请求的对象也可以根据客户应用的状态和请求类型，把请求传递给不同的处理对象。为了使这些交互简单化，所有的可能处理请求的对象应提供一致的接口。在Java中，不同处理对象可以实现一个共同的接口或者继承同一个抽象的父类来实现。

- 客户对象初始化请求，或者在不知道这些对象是否能处理这个请求的情况下初始化任何可能处理请求的对象。也就是说，客户对象和在处理链表中的处理对象都不需要知道到底哪个对象去处理这个请求。

- 请求不能保证被处理。也就是，在没有处理的情况下，请求已经到达了处理链表尾。

适用范围：

- 有多个对象可以处理一个请求，哪个对象处理该请求在运行时时刻由客户端自动确定。
- 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定时。

责任链模式的优点：

因为无法预知来自外界（客户端）的请求是属于哪种类型，每个类如果碰到它不能处理的请求只要放弃就可以。

责任链模式的缺点：

- 效率低，因为一个请求的完成可能要遍历到最后才行，当然也可以用树的概念优化。在Java Awt 1.0中，对于鼠标按键事情的处理就是使用了CoR，到Java 1.1版本以后，就使用了

Observer代替CoR。

- 扩展性差,因为在CoR中,一定要有一个统一的接口,Handler的局限性就在这里。

18.2.6 与其他模式的关系

合成模式

责任链模式的任意一环如果复杂的话都可以根据合成模式来构建其结构。

命令模式

责任链模式由于使用链式结构来解决特定需求,所以客户端无法确定是具体由哪一个对象完成的处理操作,而命令模式却明确知道其处理对象。

18.3 责任链模式总结

责任链模式的优点在上面已经体现出来了,它主要是降低了耦合、提高了灵活性。因为无法预知来自外界(客户端)的请求是属于哪种类型,每个类如果碰到它不能处理的请求只要放弃就可以。但是责任链模式可能会带来一些额外的性能损耗,因为它要从链子开头开始遍历。

在实际应用中,通常要根据需要来使用不纯的责任链模式,读者应仔细分析需求来找到使用此模式的最佳方法。

```
33 * 商品重量
34 *
35 private int weight;
36
37 /**
38 * 构造方法
39 *
40 * @param name 商品名称
41 * @param id 商品ID
42 * @param x 商品X轴坐标
43 * @param y 商品Y轴坐标
44 * @param z 商品Z轴坐标
45 * @param weight 商品重量
46 */
47 public Goods(String name, String id, int x, int y, int z, int weight) {
48     this.name = name;
49     this.id = id;
50     this.x = x;
51     this.y = y;
52     this.z = z;
53     this.weight = weight;
54 }
55
```

第19章 迭代器模式 (Iterator)

迭代器模式提供一种方法以访问一个容器 (container) 对象中的各个元素，而又不需暴露该对象的内部细节¹。

迭代器模式属于行为型模式。实际编程时，常遇到这样一种情况：一个聚合对象，比如列表，应该提供一种方法来让别人可以访问它的元素，而又不需要暴露它的内部结构。针对不同的需求，有时可能要以不同的方式遍历这个列表，但是即使可以预见所需要的所有遍历操作，可能也不希望列表的接口中充斥着各种不同的遍历操作。

在实际生活中，只要是在要管理大量的对象的情况下大部分都适合使用此模式，比如家中有很多盘子摆放在橱柜中，现在的需求是取出前五个盘子来，通常的做法是首先要有人去查看并了解盘子摆放的方式，例如盘子是竖着放还是叠着放。然后将盘子按顺序一次拿出，这种方式看似没什么问题，但是当取出的物品和物品的摆放方式经常变化的时候，这种操作就会很复杂。这时可使用迭代器模式，迭代器模式在访问集合中的对象时是十分有效的。

19.1 收银员的商品处理效率亟待提高

在超市内进行收款工作是非常辛苦的，顾客购买的商品多种多样，收款时要从购物车中取出商品，而且还需要对条形码进行扫描，所以通常收款的效率是很慢的，如图19-1所示。



图19-1 收银场景

下面展示的是通常情况下购物及收款的流程代码：

商品类：

代码片段1 Goods.java

```
1 package cn.steven.pattern.demo.iterator.question;
2
3 /**
4  * 商品
5  */
```

¹Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.GOF[95]

```

购物车 6 public class Goods {
7     /**
代码片段 8     * 商品名
9     */
10    private String name;
11
12    /**
13    * 商品条形码
14    */
15    private String id;
16
17    /**
18    * 商品宽度 m
19    */
20    private int x;
21
22    /**
23    * 商品高度 m
24    */
25    private int y;
26
27    /**
28    * 商品深度 m
29    */
30    private int z;
31
32    /**
33    * 商品重量 kg
34    */
35    private int weight;
36
37    /**
38    * 构造方法
39    *
40    * @param name
41    * @param id
42    * @param x
43    * @param y
44    * @param z
45    * @param weight
46    */
47    public Goods(String name, String id, int x, int y, int z,
48                int weight) {
49        super();
50        this.name = name;
51        this.id = id;
52        this.x = x;
53        this.y = y;
54        this.z = z;
55        this.weight = weight;
56    }
57

```



```
58     public String getName() {
59         return name;
60     }
61
62     public void setName(String name) {
63         this.name = name;
64     }
65
66     public String getId() {
67         return id;
68     }
69
70     public void setId(String id) {
71         this.id = id;
72     }
73
74     public int getX() {
75         return x;
76     }
77
78     public void setX(int x) {
79         this.x = x;
80     }
81
82     public int getY() {
83         return y;
84     }
85
86     public void setY(int y) {
87         this.y = y;
88     }
89
90     public int getZ() {
91         return z;
92     }
93
94     public void setZ(int z) {
95         this.z = z;
96     }
97
98     public int getWeight() {
99         return weight;
100    }
101
102    public void setWeight(int weight) {
103        this.weight = weight;
104    }
105 }
```

注意商品类中定义了x, y, z三个变量来表示商品的真实大小，下面为了方便起见，只使用了x值填充购物车。

购物车类:

代码片段2 Cart.java

```

1 package cn.steven.pattern.demo.iterator.question;
2
3 /**
4  * 购物车
5  */
6 public class Cart {
7
8     /**
9      * 购物车长度
10     */
11     private int length;
12
13     /**
14      * 存放空间
15     */
16     private String[] cartSpace;
17
18     /**
19      * 当前可以摆放的位置
20     */
21     private int position;
22
23     public String[] getCartSpace() {
24         return cartSpace;
25     }
26
27     public void setCartSpace(String[] cartSpace) {
28         this.cartSpace = cartSpace;
29     }
30
31     public int getLength() {
32         return length;
33     }
34
35     public void setLength(int length) {
36         this.length = length;
37     }
38
39     public int getPosition() {
40         return position;
41     }
42
43     public void setPosition(int position) {
44         this.position = position;
45     }
46
47     public Cart(int length) {
48         super();

```

```

49     this.length = length;
50     this.position = 0;
51     this.cartSpace = new String[length];
52 }
53
54 /**
55  * 存放商品
56  *
57  * @param goods 要存放的商品
58  * @return 是否成功存放
59  */
60 public boolean putGoods(Goods goods) {
61     /**
62      * 查看是否能存放
63      */
64     if (goods.getX() > (length - position)) {
65         /**
66          * 体积太大
67          */
68         System.out.println("超出容积");
69         return false;
70     } else {
71         for (int i = 0; i < goods.getX(); i++) {
72             cartSpace[position] = goods.getName();
73             position++;
74         }
75         return true;
76     }
77 }
78
79 }

```

注意到购物车代码中使用了一个数组作为其空间，当目前已经放不下指定商品时会显示“超出容积”字样，默认情况下采用顺序存放的方法。

客户使用类：

代码片段3 Client.java

```

1 package cn.steven.pattern.demo.iterator.question;
2
3 /**
4  * 购物客户使用的代码
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 构建购物车
12          */
13         Cart cart = new Cart(10);
14

```



```

15 public abstract class Aggregate {
16     /**
17      * 填充物品
18      */
19     cart.putGoods(new Goods("电视", "001", 2, 2, 4, 15));
20     cart.putGoods(new Goods("冰箱", "004", 7, 1, 3, 25));
21     cart.putGoods(new Goods("杯子", "055", 1, 1, 1, 4));
22     cart.putGoods(new Goods("笔筒", "063", 1, 1, 1, 4));
23
24     System.out.println("查看购物车: [容积: " + cart.getLength() + "I");
25     for (int i = 0; i < cart.getLength(); i++) {
26         System.out.println((i + 1) + ": " + cart.getCartSpace()[i]);
27     }
28     System.out.println();
29
30     /**
31      * 收款过程
32      */
33     String goodsName = null;
34     for (int i = 0; i < cart.getLength(); i++) {
35         if (cart.getCartSpace()[i] != null) {
36             if (!cart.getCartSpace()[i].equals(goodsName)) {
37                 System.out.println("收款: " + cart.getCartSpace()[i]);
38                 goodsName = cart.getCartSpace()[i];
39             } else {
40                 continue;
41             }
42         }
43     }
44 }
45
46 }

```

客户使用代码中使用了如下步骤来操作购物车:

- 创建购物车。
- 购买商品 (有可能超出容积)。
- 查看购物车状态。
- 遍历购物车结账。

运行结果如图19-2所示。

由上述结果可见, 通过这种直观的编程方式的确可以解决收款的问题, 但是仔细观察可以发现, 代码 (也就是操作过程) 十分复杂 (见代码片段1中第33行~第42行), 而且不具备通用性, 比如购物车的形状改变或改用其他的容器时, 收款人就需要更换另一种方式进行货物的清点。

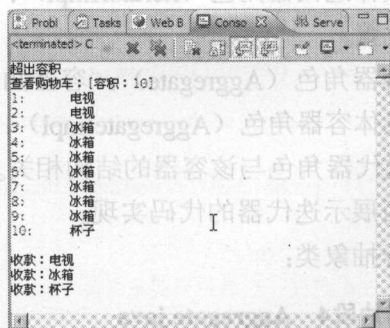


图19-2 购物运行结果

总体来说以上实现的缺点如下：

- 收款员必须了解购物车的细节才能结算其中的商品。
- 收款员不方便使用多种方式获取购物车商品。
- 多种购物车和商品的摆放方式极大地降低了收款员的效率。

所以，连锁店的工作效率如果需提高，需要从收款员的具体工作开始改造。

19.2 迭代器模式的结构

迭代器模式提供给外部一个良好的操作集合的办法，这种办法避免了暴露集合的细节，外部的使用代码也将更加简单。

19.2.1 迭代器模式

迭代器将具体的集合类和迭代类分离开来，使迭代器和具体集合类的复用性更佳，且外界代码使用时可以使用迭代器遍历集合，从而无需关心集合的内部细节，其类图如图19-3所示。

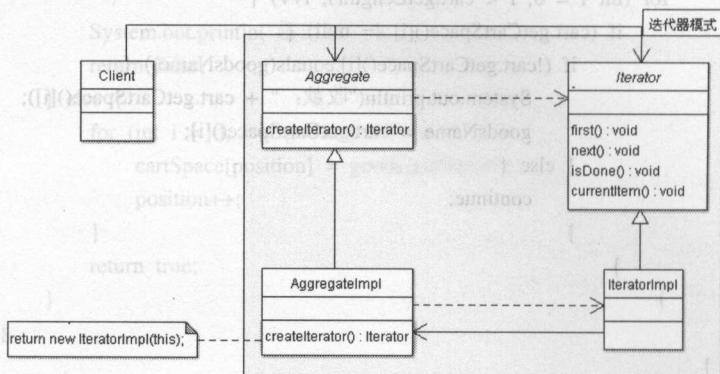


图19-3 迭代器模式类图

其中的参与者如下：

- 迭代器角色（Iterator）：迭代器角色负责定义访问和遍历元素的接口。
- 具体迭代器角色（IteratorImpl）：具体迭代器角色要实现迭代器接口，并要记录遍历中的当前位置。
- 容器角色（Aggregate）：容器角色负责提供创建具体迭代器角色的接口。
- 具体容器角色（AggregateImpl）：具体容器角色实现创建具体迭代器角色的接口——这个具体迭代器角色与该容器的结构相关。

下面展示迭代器的代码实现。

集合抽象类：

代码片段4 Aggregate.java

```
1 package cn.steven.pattern.demo.iterator.pattern;
2
3 /**
4  * 集合抽象类
5  */
```

```

6 public abstract class Aggregate {
7     /**
8      * 创建迭代器
9      * 构造方法
10     * @return
11     */
12     public abstract Iterator createIterator();
13 }

```

迭代器抽象类:

代码片段5 Iterator.java

```

1 package cn.steven.pattern.demo.iterator.pattern;
2
3 /**
4  * 迭代器抽象类
5  */
6 public abstract class Iterator {
7
8     /**
9      * 移动至集合头部
10     */
11     public abstract void first();
12
13     /**
14      * 移动至下一个元素
15     */
16     public abstract void next();
17
18     /**
19      * 是否已达集合尾部
20     */
21     public abstract boolean isDone();
22
23     /**
24      * 返回当前元素
25     */
26     public abstract Object currentItem();
27
28 }

```

具体集合类:

代码片段6 AggregateImpl.java

```

1 package cn.steven.pattern.demo.iterator.pattern;
2
3 public class AggregateImpl extends Aggregate {
4     /**
5      * 一个集合
6      */
7
8 }

```

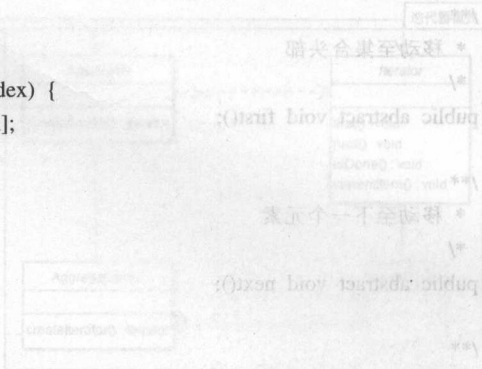


```
8 private int[] aIntList;
9
10 /**
11  * 构造方法
12  */
13 public AggregateImpl() {
14     aIntList = new int[] { 3, 4, 6, 2, 9 };
15 }
16
17 /**
18  * 得到集合大小
19  *
20  * @return
21  */
22 public int getSize() {
23     return aIntList.length;
24 }
25
26 /**
27  * 取得集合元素
28  */
29 public int getItem(int index) {
30     return aIntList[index];
31 }
32
33 /**
34  * 创建迭代器
35  *
36  * @return
37  */
38 @Override
39 public Iterator createIterator() {
40     return new IteratorImpl(this);
41 }
42 }
```

具体迭代器类:

代码片段7 IteratorImpl.java

```
1 package cn.steven.pattern.demo.iterator.pattern;
2
3 public class IteratorImpl extends Iterator {
4
5     /**
6      * 维护当前索引号
7      */
8     private int index;
9
10    /**
11     * 集合
12     */
```



具体迭代器角色 (IteratorImpl)：具体迭代器角色负责实现迭代器接口，并要记录遍历中的当前元素索引。容器角色负责提供创建迭代器角色的接口——这个接口就是迭代器接口。具体迭代器角色与容器的结构关系如下面展示迭代器的代码实现。

代码片段14 AggregateImpl.java

```
1 package cn.steven.pattern.demo.iterator.pattern;
2
3 public class AggregateImpl extends Aggregate {
4
5     /**
6      * 集合
7      */
```

```
13 private AggregateImpl aggregateImpl;
14
15 /**
16  * 构造方法
17  */
18 public IteratorImpl(AggregateImpl aggregateImpl) {
19     this.aggregateImpl = aggregateImpl;
20     first();
21 }
22
23 @Override
24 public Object currentItem() {
25     return aggregateImpl.getItem(index);
26 }
27
28 @Override
29 public void first() {
30     index = 0;
31 }
32
33 @Override
34 public boolean isDone() {
35     /**
36      * 如果超出索引范围则执行isDone
37      */
38     if (index >= aggregateImpl.getSize()) {
39         return true;
40     } else {
41         return false;
42     }
43 }
44
45 @Override
46 public void next() {
47     if (index < aggregateImpl.getSize()) {
48         index++;
49     }
50 }
51
52 }
```

以上代码可见，迭代器模式将集合数据和集合的遍历方式完全隔离开，起到了很好的解耦合作用。

客户端代码：

代码片段8 Client.java

```
1 package cn.steven.pattern.demo.iterator.pattern;
2
3 /**
4  * 迭代器客户端
5  */
```

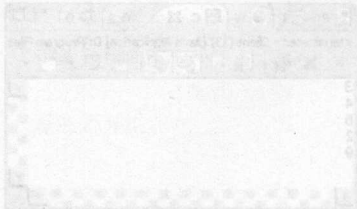


图 19-4 迭代器模式类图

```
6 public class Client {
7
8     public static void main(String[] args) {
9         /**
10          * 创建集合
11          */
12         Aggregate aggregate = new AggregateImpl();
13
14         /**
15          * 创建迭代器
16          */
17         Iterator iterator = aggregate.createIterator();
18
19         /**
20          * 访问集合
21          */
22         for(;!iterator.isDone();iterator.next()){
23             System.out.println(iterator.currentItem());
24         }
25     }
26 }
27 }
```

注意代码片段8第22行使用了迭代器遍历的方法。代码运行结果如图19-4所示。

对比代码片段3和代码片段8可见，通过迭代器的方式遍历集合，极大地方便了客户代码的使用，且可以通过增加迭代器来实现不同的迭代方法，比如带排序功能的迭代，也可以加入删除、修改、插入等功能，但是迭代器加入修改数据的功能通常会有很多的限制。通常来说，使用迭代器来进行集合中的元素访问是最常见的。

19.2.2 用迭代器模式统一处理各类商品

在本章第一节中，我们已经讨论过收银员商品处理效率低下的问题。通过上一节的学习，我们应该已经初步掌握了迭代器模式的设计和使用方法，在本节中，将会继续扩展其应用。下面我们的目的不只是将收银员的工作量降低，而且还要扩展带排序功能的迭代器。

设计类图如图19-5所示。

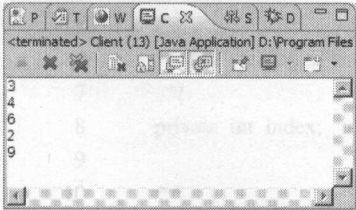


图19-4 迭代器运行结果

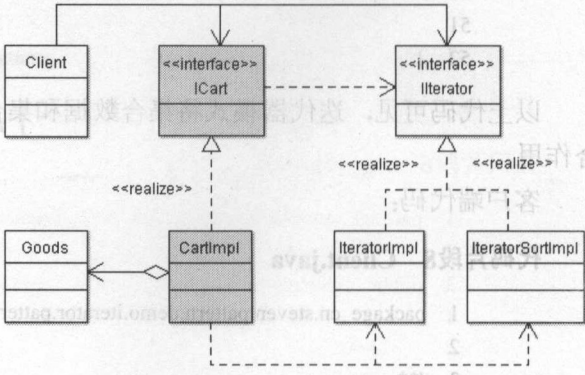


图19-5 商品处理类图

在本例中，将会使用两个不同功能的迭代器对同一个集合进行迭代，代码展示如下。

商品类（简化版）：

代码片段9 Goods.java

```
1 package cn.steven.pattern.demo.iterator;
2
3 /**
4  * 商品类
5  */
6 public class Goods {
7     /**
8      * 商品名称
9      */
10    private String name;
11
12    /**
13     * 商品价格
14     */
15    private float price;
16
17    public String getName() {
18        return name;
19    }
20
21    public void setName(String name) {
22        this.name = name;
23    }
24
25    public float getPrice() {
26        return price;
27    }
28
29    public void setPrice(float price) {
30        this.price = price;
31    }
32
33    /**
34     * 构造方法
35     *
36     * @param name
37     * @param price
38     */
39    public Goods(String name, float price) {
40        super();
41        this.name = name;
42        this.price = price;
43    }
44
45 }
```

购物车接口：

代码片段10 ICart.java

```
1 package cn.steven.pattern.demo.iterator;
2
3 /**
4  * 购物车接口
5  */
6 public interface ICart {
7
8     /**
9      * 普通迭代器
10     */
11     Iterator createIterator();
12
13     /**
14      * 排序迭代器
15     */
16     Iterator createSortIterator();
17 }
```

注意此接口中声明了两个迭代器，所以具备两种迭代功能。
具体购物车：

代码片段11 CartImpl.java

```
1 package cn.steven.pattern.demo.iterator;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 /**
7  * 购物车类
8  */
9 public class CartImpl implements ICart {
10
11     private List<Goods> goodsList;
12
13     /**
14      * 得到集合大小
15      *
16      * @return
17      */
18     public int getSize() {
19         return goodsList.size();
20     }
21
22     /**
23      * 取得集合元素
24      */
25     public Goods getItem(int index) {
26         return goodsList.get(index);
27     }
```

```

28     index = 0;
29     /**
30     * 构造方法
31     */
32     public CartImpl() {
33         goodsList = new LinkedList<Goods>();
34     }
35
36     /**
37     * 加入元素方法
38     */
39     public void add(Goods goods) {
40         goodsList.add(goods);
41     }
42
43     /**
44     * 迭代器方法
45     */
46     @Override
47     public Iterator createIterator() {
48         return new IteratorImpl(this);
49     }
50
51     /**
52     * 迭代器方法
53     */
54     @Override
55     public Iterator createSortIterator() {
56         return new IteratorSortImpl(this);
57     }
58
59 }

```

代码片段11中的设计比起代码片段2的设计简易了很多，主要是因为采用了JDK提供的集合类来存放数据，而原来是使用原始的数组进行存放。

迭代器接口：

代码片段12 Iterator.java

```

1  package cn.steven.pattern.demo.iterator;
2
3  /**
4   * 迭代器接口
5   */
6  public interface Iterator {
7      /**
8       * 移动至集合头部
9       */
10     void first();
11
12     /**
13     * 移动至下一个元素

```



```

14      */
15      void next();
16
17      /**
18       * 是否已达集合尾部
19       */
20      boolean isDone();
21
22      /**
23       * 返回当前元素
24       */
25      Goods currentItem();
26  }

```

普通的迭代器实现:

代码片段13 IteratorImpl.java

```

1  package cn.steven.pattern.demo.iterator;
2
3  /**
4   * 普通迭代器
5   */
6  public class IteratorImpl implements Iterator {
7
8      /** cn.steven.pattern.demo.iterator
9       * 当前索引
10      */
11      private int index;
12
13      /**
14       * 集合
15       */
16      private CartImpl cartImpl;
17
18      /**
19       * 构造方法
20       *
21       * @param cartImpl
22       */
23      public IteratorImpl(CartImpl cartImpl) {
24          this.cartImpl = cartImpl;
25          first();
26      }
27
28      @Override
29      public Goods currentItem() {
30          return cartImpl.getItem(index);
31      }
32
33      @Override
34      public void first() {

```



```

25      /**
26      * 复制集合
27      */
28      goodsArray = new Goods[cartImpl.getSize()];
29      for (int i = 0; i < cartImpl.getSize(); i++) {
30          goodsArray[i] = cartImpl.getItem(i);
31      }
32
33      /**
34      * 排序
35      */
36      sort();
37
38      first();
39  }
40
41  /**
42  * 冒泡排序
43  */
44  public void sort() {
45      for (int i = 0; i < goodsArray.length - 1; i++) {
46          for (int j = 0; j < goodsArray.length - i - 1; j++) {
47              /**
48              * 按照金额从小到大排列
49              */
50              if (goodsArray[j].getPrice() > goodsArray[j + 1]
51                  .getPrice()) {
52                  Goods temp = goodsArray[j + 1];
53                  goodsArray[j + 1] = goodsArray[j];
54                  goodsArray[j] = temp;
55              }
56          }
57      }
58  }
59
60  @Override
61  public Goods currentItem() {
62      return goodsArray[index];
63  }
64
65  @Override
66  public void first() {
67      index = 0;
68  }
69
70  @Override
71  public boolean isDone() {
72      /**
73      * 如果超出索引范围则执行isDone
74      */
75      if (index >= goodsArray.length) {

```



```

76         return true;
77     } else {
78         return false;
79     }
80 }
81
82 @Override
83 public void next() {
84     if (index < goodsArray.length) {
85         index++;
86     }
87 }
88
89 }

```

注意代码片段14中第44行~第58行采用了冒泡排序 (Bubble Sort)¹的方式进行排序。
客户端代码:

代码片段15 Client.java

```

1 package cn.steven.pattern.demo.iterator;
2
3 /**
4  * 迭代器模式客户端
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 创建集合
12          */
13         CartImpl cart = new CartImpl();
14
15         /**
16          * 加入数据
17          */
18         cart.add(new Goods("电视机", 3200.99f));
19         cart.add(new Goods("矿泉水", 0.8f));
20         cart.add(new Goods("笔记本", 5200f));
21         cart.add(new Goods("茶壶", 22.99f));
22         cart.add(new Goods("口香糖", 5.98f));
23
24         /**
25          * 测试普通迭代器
26          */
27         Iterator iterator = cart.createIterator();
28         System.out.println("==普通迭代==");
29         for (; !iterator.isDone(); iterator.next()) {

```

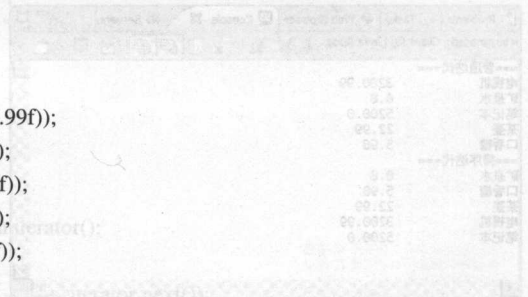


图 19-6 购物车界面

¹<http://zh.wikipedia.org/wiki/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F>.

```
30         Goods g = iterator.currentItem();
31         System.out.println(g.getName() + "\t\t" + g.getPrice());
32     }
33
34     /**
35      * 测试排序迭代器
36      */
37     Iterator iterator2 = cart.createSortIterator();
38     System.out.println("===排序迭代===");
39     for (; iterator2.isDone(); iterator2.next()) {
40         Goods g = iterator2.currentItem();
41         System.out.println(g.getName() + "\t\t" + g.getPrice());
42     }
43
44     System.out.println();
45 }
46
47 }
```

在代码片段15中使用了两种迭代器分别进行处理，这样就给了客户端更大的自由度，而且还可以根据需求编写其他功能的迭代器。

客户端代码运行结果如图19-6所示。

由以上结果可见，普通迭代和排序迭代（按照价格升序）都可以成功运行了，反观客户端代码可见使用不同的迭代器的代码几乎完全一样，这样就保证了用户接口的稳定生和易用性。

19.2.3 迭代器模式在JDK中的实例

JDK中已经设计了良好的迭代器模式，其主要集中在java.util的集合包中。下面是迭代器的结构类，如图19-7所示。

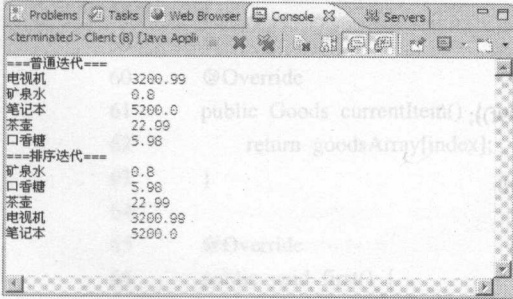


图19-6 商品处理结果

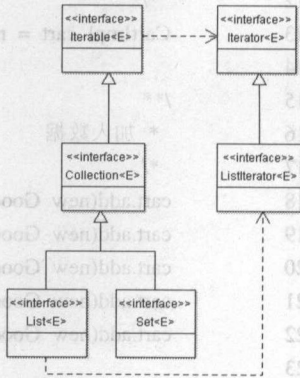


图19-7 集合接口（部分）

由图19-7可见，集合体系中已经默认集成了迭代器模式，常用的List接口和Set接口可以方便地使用Iterator，而且List接口还可以生成一个ListIterator接口，此接口具有比Iterator更多的功能。

同为集合的Map接口不在此体系内，但是由于Map是由key和value组成的，而key具备Set接口的特性，所以也可以间接使用Iterator来遍历。

下面展示的代码作用为遍历一个集合的各种方式:

代码片段16 UseCollection.java

```
1 package cn.steven.pattern.demo.iterator.jdk;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5 import java.util.Iterator;
6 import java.util.List;
7 import java.util.Map;
8 import java.util.TreeMap;
9
10 /**
11  * 集合的使用方法
12  */
13 public class UseCollection {
14
15     public static void main(String[] args) {
16
17         /**
18          * 创建集合
19          */
20         Collection<String> collection = new ArrayList<String>();
21         int size = 3;
22         for (int i = 0; i < 3; i++) {
23             collection.add("item " + i);
24         }
25
26         /**
27          * 索引遍历法, 需要转换为List接口
28          */
29         List<String> list = (List<String>) collection;
30         for (int i = 0; i < list.size(); i++) {
31             System.out.println("索引遍历: " + list.get(i));
32         }
33
34         /**
35          * 迭代器遍历
36          */
37         Iterator<String> iterator = collection.iterator();
38         while (iterator.hasNext()) {
39             System.out.println("迭代器遍历: " + iterator.next());
40         }
41
42         /**
43          * 增强for循环, JDK1.5+
44          */
45         for (String s : collection) {
46             System.out.println("增强for遍历: " + s);
47         }
48     }
49 }
```



```

49      /**
50       * 构建Map
51       */
52      Map<String, String> m = new TreeMap<String, String>();
53
54      /**
55       * 填充数据
56       */
57      m.put("西施", "春秋");
58      m.put("王昭君", "西汉");
59      m.put("貂蝉", "东汉");
60      m.put("杨玉环", "唐朝");
61
62      /**
63       * Set迭代器遍历
64       */
65      Iterator<String> setIterator = m.keySet().iterator();
66      while (setIterator.hasNext()) {
67          String k = setIterator.next();
68          System.out.println("set迭代器遍历: " + k + " 在 " + m.get(k));
69      }
70
71      /**
72       * 增强for循环, JDK1.5+
73       */
74      for (Map.Entry<String, String> e : m.entrySet()) {
75          System.out.println("增强for循环: " + e.getKey() + " 在 "
76                          + e.getValue());
77      }
78
79  }
80
81  }

```

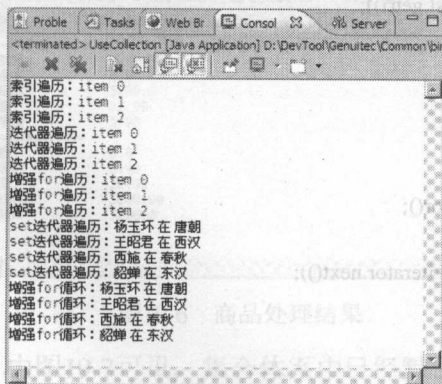


图19-8 集合使用运行结果

运行结果如图19-8所示。

上面遍历集合的例子中使用了多种方式,一般推荐使用增强的for式语法,但是需要JDK1.5+¹的版本支持,如果版本低于1.5建议采用迭代器式语法。

在旧版本的JDK中还有一种迭代方式使用了Enumeration接口,其描述如下:

public interface Enumeration<E>

它实现Enumeration接口的对象,生成一系列元素,一次生成一个。连续调用nextElement方法将返回一系列的连续元素。

¹<http://java.sun.com/j2se/1.5.0/docs/guide/language/foreach.html>.

例如, 要输出 `Vector<E> v` 的所有元素, 可使用以下方法:

```
for (Enumeration<E> e = v.elements(); e.hasMoreElements();  
    System.out.println(e.nextElement());
```

这些方法主要通过向量的元素、哈希表的键以及哈希表中的值进行枚举。枚举也用于将输入流指定到 `SequenceInputStream` 中。

注意: 此接口的功能与 `Iterator` 接口的功能是重复的。此外, `Iterator` 接口添加了一个可选的移除操作, 并使用较短的方法名。新的实现应该优先考虑使用 `Iterator` 接口而不是 `Enumeration` 接口。

从以下版本开始: `JDK 1.0`。

注意查看说明中的注意部分, 此接口目前已经不推荐使用了。

19.2.4 迭代器模式的使用范围及优点

迭代器模式的优点:

- 支持以不同的方式遍历一个容器角色。根据实现方式的不同, 效果上会有差别。
- 简化了容器的接口。但是在 `Java Collection` 中为了提高可扩展性, 容器还是提供了遍历的接口。
- 对同一个容器对象, 可以同时进行多个遍历。因为遍历状态是保存在每一个迭代器对象中的。

迭代器模式的缺点: 聚合密切相关, 增加了耦合。但将耦合定义在抽象基类中, 可解决这个问题, 具体方法可参考 `JDK` 的实现。

使用情景:

- 需要遍历访问聚集中的对象而不能暴露聚集的内部结构时。
- 允许对聚集的多级遍历访问而不会相互受影响时。
- 要提供一个一致的接口来遍历访问聚集中不同的结构时。

19.2.5 与其他模式的关系

如果有一个组合模式的结构, 通常可以使用迭代器模式来遍历。

迭代器内部通常会需要维护一些集合的相关数据状态, 可以使用备忘录模式存储其状态。

19.3 外观模式总结

迭代器模式 (`Iterator`) 提供了一种方法可以顺序访问一个聚合对象中的元素, 而不暴露该集合对象的内部表示。迭代器模式分离了集合对象的遍历行为, 抽象出一个迭代器类来进行负责, 这样既可以不暴露集合的内部机构, 又可以让外部代码透明地访问集合内部的数据, 这种方式很好地将迭代操作和迭代实现分离开来, 使客户端代码的编写极大地被简化了。

如果在需求中有类似操作一个集合数据的需求, 可以考虑使用迭代器模式。

第20章 访问者模式 (Visitor)

访问者模式定义一个作用于某对象结构中的各元素上的操作，它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作¹。

系统中一个已经完成的类层次结构，通常已经对其提供了满足需求的接口。但是面对新增的需求，又如何满足呢？如果这是为数不多的几次变动，而且不用为了一个需求的调整而将整个类层次结构全修改一遍，那么直接在原有类层次结构上修改也许是个不错的主意。

但是实际往往遇到的却是：这样的需求变动也许会不停发生，如果每次都修改类结构，则最终的类结构会发生翻天覆地的变化。

对于类的使用者来说，这更是一个重大的事情，因为类的结构更改了，这样以前使用类的代码可能无法继续使用了。而面向对象的原则——变化要依赖不变，也就成了空谈。

本章介绍的访问者模式就是用来在不更改类结构的情况下为类增加功能的模式。

20.1 再谈收银员的效率问题

在超市内进行收款是一件非常辛苦的工作，因为收款时要将物品依次取出，浪费了大量的时间，这个问题已经在迭代器模式中进行解决了，但是收款的过程依然很慢。

其问题在于某些商品有可能要附加其他的操作，比如食品要放入保温盒中，玩具要放在盒子中，再比如，每一类的商品可能折扣价格是不同的，需要单独计算。

按照通常的编程方法可以有如下代码：

产品抽象类：

代码片段1 Goods.java

```
1 package cn.steven.pattern.demo.visitor.quest;
2
3 /**
4  * 商品
5  */
6 public abstract class Goods {
7     /**
8      * 商品名
9      */
10    private String name;
11
12    /**
13     * 价格
14     */
15    private float price;
```

¹Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.GOF[95]


```
16 public String getName() {
17     return name;
18 }
19
20 public void setName(String name) {
21     this.name = name;
22 }
23
24 public float getPrice() {
25     return price;
26 }
27
28 public void setPrice(float price) {
29     this.price = price;
30 }
31
32 }
```

食品类:

代码片段2 FoodGoods.java

```
1 package cn.steven.pattern.demo.visitor.quest;
2
3 /**
4  * 食物商品
5  */
6 public class FoodGoods extends Goods {
7
8     /**
9      * 构造方法
10      *
11      * @param name 商品名
12      */
13     public FoodGoods(String name, float price) {
14         this.setName(name);
15         this.setPrice(price);
16     }
17 }
```

玩具类:

代码片段3 ToyGoods.java

```
1 package cn.steven.pattern.demo.visitor.quest;
2
3 /**
4  * 玩具商品
5  */
6 public class ToyGoods extends Goods {
7
8     /**
9      * 构造方法
```

```

10      *
11      * @param name 商品名
12      */
13      public ToyGoods(String name, float price) {
14          this.setName(name);
15          this.setPrice(price);
16      }
17
18  }

```

客户端代码:

代码片段4 Client.java

```

1  package cn.steven.pattern.demo.visitor.quest;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * 客户端代码
8   */
9  public class Client {
10
11      public static void main(String[] args) {
12          /**
13           * 购物车
14           */
15          List<Goods> cart = new ArrayList<Goods>();
16          cart.add(new FoodGoods("烤鸭", 108f));
17          cart.add(new ToyGoods("葫芦娃", 37.9f));
18          cart.add(new FoodGoods("凉粉", 5.5f));
19          cart.add(new ToyGoods("喜羊羊", 37.9f));
20
21          /**
22           * 操作
23           * 若此处要操作一个组合模式的对象，是很不方便的
24           * 具体的组合模式对象参见本章模式中的例子
25           */
26          for (Goods goods : cart) {
27
28              if (goods instanceof FoodGoods) {
29                  // 食物需要用保温盒包装
30                  System.out.println("用保温盒包装 "
31                      + goods.getName() + " 收款:"
32                      + goods.getPrice());
33              } else if (goods instanceof ToyGoods) {
34                  // 玩具用纸盒包装
35                  System.out.println("用纸盒包装 "
36                      + goods.getName() + " 收款:"
37                      + goods.getPrice());
38              } else {
39                  // 其他商品不做处理

```

```

39 abstract class Fruit { System.out.println(goods.getName() + " 收款:"
40     private String name; + goods.getPrice());
41     }
42     public String getName() { return name; }
43 }
44 }

```

注意代码中所使用的instanceof运算符¹的功能是判读一个对象是否符合一个类/接口。客户端代码的运行结果如图20-1所示。

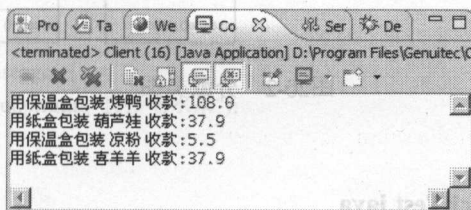


图20-1 运行结果

由图20-1可见，如上的代码的确可以满足对特定对象进行特定操作的办法，但是扩展性非常不好，试想如果要增加新的类对象，势必要在if语句中增加新对象的判断，这样就影响了客户代码的稳定性。

本章介绍的访问者模式可以将这种影响通过特定的编程方法加以化解。

20.2 访问者模式的结构

访问者模式将类结构和对类增加的功能解耦，使得客户使用和编程更加方便。

20.2.1 静态、动态、单分派、多分派、双重分派

在Java中实现访问者模式需要使用双重分派²，为了理解这个名词必须要从静态、动态、单分派、多分派这几个名词谈起，初次接触这些名词的读者比较容易混淆其概念，下面将使用实例来说明其含义³。

静态类型：变量被声明时的类型是静态类型。

动态类型：变量所引用的对象的真实类型。

宗量：方法所属的对象和方法的参数的统称。

单分派：根据一个宗量的类型进行方法的选择。

多分派：根据多于一个宗量的类型进行方法的选择。

多重分派：采用多次单分派的方式进行方法选择。

双重分派：采用两次单分派的方式进行方法选择。

首先看一下静态分派的案例，类图如图20-2所示。

¹<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/op2.html>.

²http://en.wikipedia.org/wiki/Double_dispatch.

³http://en.wikipedia.org/wiki/Dynamic_dispatch.

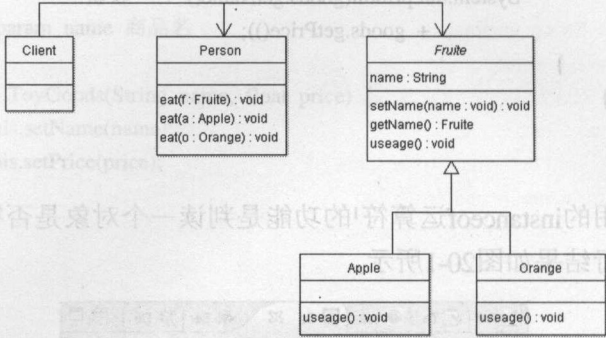


图20-2 分派类图

代码如下：

代码片段5 StaticDispatchTest.java

```
1 package cn.steven.pattern.demo.visitor.doubledispatch;
2
3 /**
4  * 方法重载的静态分派
5  */
6 public class StaticDispatchTest {
7
8     public static void main(String[] args) {
9
10         Person p = new Person();
11
12         /**
13          * 编译时就被确定其类型
14          * 所以匹配正确
15          */
16         Apple x = new Apple();
17         Orange y = new Orange();
18         p.eat(x);
19         p.eat(y);
20
21         /**
22          * 编译时就被确定其类型
23          * 所以匹配的是 public void eat(Fruit fruit)
24          */
25         Fruit a = new Apple();
26         Fruit b = new Orange();
27         p.eat(a);
28         p.eat(b);
29     }
30
31 }
32
33 /**
34  * 抽象水果类
35  */
```

```

36 abstract class Fruit {
37     private String name;
38
39     public String getName() {
40         return name;
41     }
42
43     public void setName(String name) {
44         this.name = name;
45     }
46
47     public void useage(){
48         System.out.println("直接吃水果");
49     }
50 }
51
52 /**
53  * 苹果
54  */
55 class Apple extends Fruit {
56     public Apple() {
57         super();
58         this.setName("苹果");
59     }
60
61     public void useage(){
62         System.out.println("洗洗吃苹果");
63     }
64 }
65
66 /**
67  * 橘子
68  */
69 class Orange extends Fruit {
70     public Orange() {
71         super();
72         this.setName("橘子");
73     }
74
75     public void useage(){
76         System.out.println("剥皮吃橘子");
77     }
78 }
79
80 /**
81  * 人，方法重载
82  */
83 class Person {
84     public void eat(Fruit fruit) {
85         System.out.println("吃水果");
86     }
87

```

```
88     public void eat(Apple fruit) {
89         System.out.println("吃苹果");
90     }
91
92     public void eat(Orange fruit) {
93         System.out.println("吃橘子");
94     }
95 }
```

上例是典型的静态多分派的例子。“静态”的变量参见代码片段5的第16行、17行、25行、26行，多分派参见代码片段5第18行、19行、27行、28行，此处的p变量和x、y、a、b就构成了多个“宗量”，所以结果如图20-3所示。

下面是动态单分派的代码：

代码片段6 DynamicDispatch.java

```
1  package cn.steven.pattern.demo.visitor.doubledispatch;
2
3  /**
4   * 方法重写，动态分派
5   */
6  public class DynamicDispatch {
7      public static void main(String[] args) {
8          Apple a = new Apple();
9          a.useage();
10         Fruit f = new Apple();
11         f.useage();
12     }
13 }
```

动态指的是代码片段6中第8行、10行的变量应用在第9行、第11行时的特性，因为第9行、第11行在选择方法时只使用了一个宗量，所以这时是单分派。代码的运行结果如图20-4所示。

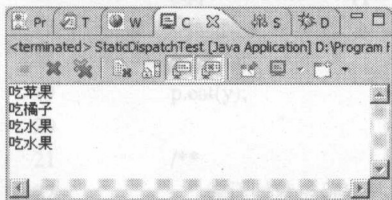


图20-3 静态多分派结果

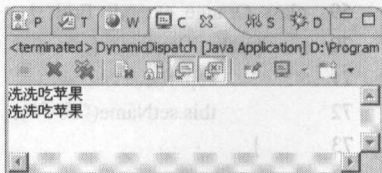


图20-4 动态单分派

由上述例子可以得出如下结论：**在Java中支持动态单分派和静态多分派，不支持动态多分派。**

静态分派的参数类型是编程时已确定的，不支持在运行时刻动态确定其参数类型，所以一定程度上不满足多态的要求，如果想要Java支持动态多分派，就必须执行两次动态的单分派，这种编程方式称为双重分派，代码如下所示：

代码片段7 Dispatch.java

```
1  package cn.steven.pattern.demo.visitor.doubledispatch;
2
```



```

3 import java.util.Date;
4
5 /**
6  * 测试Java的分派特性
7  */
8 public class Dispatch {
9     public static void main(String[] args) {
10
11         Planet p = new Planet();
12         Planet e = new Earth();
13
14         /**
15          * 静态多分派 static-multi-dispatch
16          */
17         System.out.println("静态多分派 static-multi-dispatch");
18         e.show(new String("ball"));
19         e.show((Object) new String("ball yet"));
20         e.show(new Date()); // 此处将视为Object参数
21
22         /**
23          * 动态单分派 dynamic-uni-dispatch
24          */
25         System.out.println("动态单分派 dynamic-uni-dispatch");
26         p.turn();
27         e.turn();
28
29         /**
30          * 多重分派模拟动态多分派
31          */
32         System.out.println("多重分派 multiple dispatch 模拟动态多分派 ");
33         p.show(new ShowerImpl());
34         e.show(new ShowerImpl());
35     }
36 }
37
38 class Planet {
39     /**
40      * 双重分派
41      */
42     public void show(Shower s) {
43         s.show(this);
44     }
45
46     public void show(String msg) {
47         System.out.println("Planet String as " + msg);
48     }
49
50     public void show(Object obj) {
51         System.out.println("Planet Object as " + obj);
52     }
53 }

```

```
54
55     public void turn() {
56         System.out.println("Planet turn");
57     }
58 }
59
60 class Earth extends Planet {
61     public void turn() {
62         System.out.println("Earth turn");
63     }
64 }
65
66 abstract interface Shower {
67     void show(Planet p);
68 }
69
70 class ShowerImpl implements Shower {
71     @Override
72     public void show(Planet p) {
73         System.out.println("show: " + p.getClass());
74     }
75 }
```

注意，代码片段7中增加了一个接口，一个实现类，Planet中增加了一个方法用于进行双重分派。代码运行结果如图20-5所示。

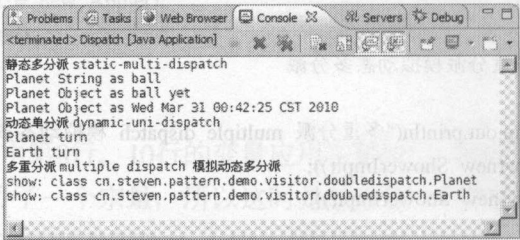


图20-5 双重分派

理解双重分派的概念是学习访问者模式的基础，以上案例请读者多进行思考练习，然后再学习模式本身。对于本身即支持动态多分派的语言（CLOS¹、Cecil²）来说，此种编程方式是没有必要的。

20.2.2 访问者模式

为了描述访问者模式的具体实现方式，下面使用了一个“见人问好”的例子，比如说在公司中有两种人，一种是领导，向其问好时要说“领导好”，另一种是普通员工，向其问好时说“你好”，人们通常需要在“运行”时刻动态地分辨出对方身份并进行问好。另一个需求是员工的工作方法，要向领导汇报工作，要与员工合作开展工作。这两个需求都必须在不改变人员类的基础上进行增加功能。

¹http://en.wikipedia.org/wiki/Common_Lisp_Object_System。
²http://en.wikipedia.org/wiki/Cecil_%28programming_language%29。

使用访问者模式必须构建至少两个类层次，一个是访问者类层次，一个是被访问的元素类层次，访问者模式的类图如图20-6所示。

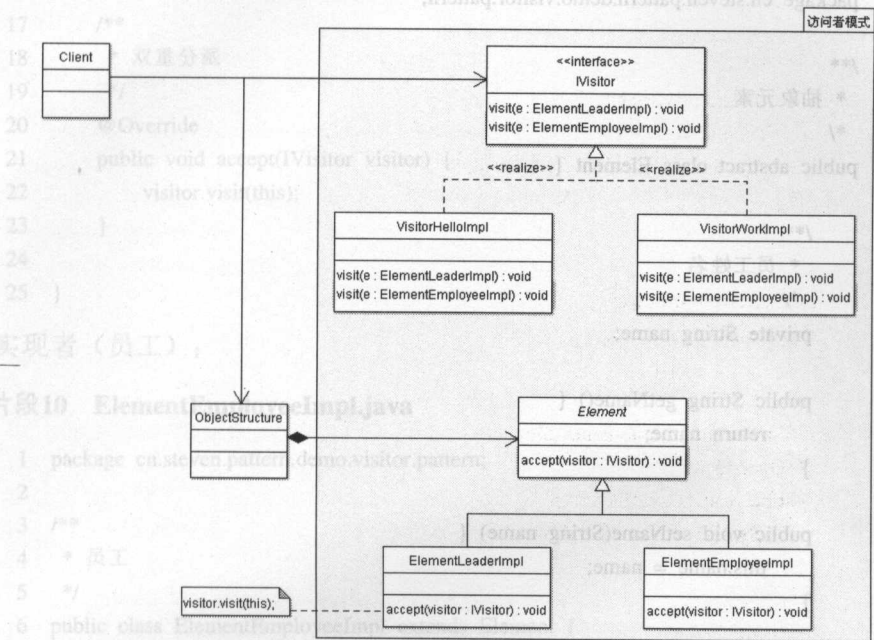


图20-6 访问者模式

- 图中的参与者如下：
- 抽象访问者（IVisitor）：声明一个或者多个访问操作，形成所有的具体元素都要实现的接口。
 - 具体访问者（VisitorXXXImpl）：实现抽象访问者所声明的接口。
 - 抽象节点（Element）：声明一个接受操作，接受一个访问者对象作为参量。
 - 具体节点（ElementXXXImpl）：实现了抽象元素所规定的接受操作。
 - 结构对象（ObjectStructure）：遍历结构中的所有元素，类似List、Set等。
- 运行时的序列图如图20-7所示。

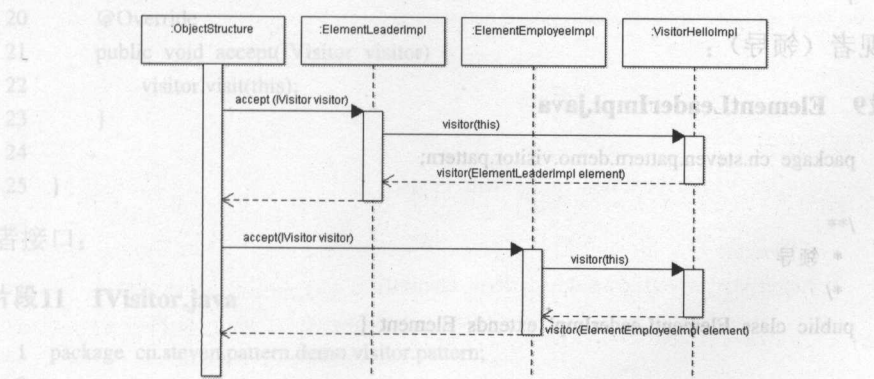


图20-7 访问者模式序列图

由图20-7中可以很容易地看到，双重分派的顺序首先是ObjectStructure到Element的子类，然后再分派到IVisitor的实现类，这样就完成了双重分派。

抽象元素类如下：

代码片段8 Element.java

```
1 package cn.steven.pattern.demo.visitor.pattern;
2
3 /**
4  * 抽象元素
5  */
6 public abstract class Element {
7
8     /**
9      * 员工姓名
10     */
11     private String name;
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     /**
22      * 构造方法
23      */
24     public Element(String name) {
25         super();
26         this.name = name;
27     }
28
29     /**
30      * 接受访问者方法，双重分派
31     */
32     public abstract void accept(IVisitor visitor);
33 }
```

元素实现者（领导）：

代码片段9 ElementLeaderImpl.java

```
1 package cn.steven.pattern.demo.visitor.pattern;
2
3 /**
4  * 领导
5  */
6 public class ElementLeaderImpl extends Element {
7
8     /**
9      * 构造方法
10     */
11     @param name
12     */
```

```
13 public ElementLeaderImpl(String name) {
14     super(name);
15 }
16
17 /**
18  * 双重分派
19  */
20 @Override
21 public void accept(IVisitor visitor) {
22     visitor.visit(this);
23 }
24
25 }
```

元素实现者 (员工) :

代码片段10 ElementEmployeeImpl.java

```
1 package cn.steven.pattern.demo.visitor.pattern;
2
3 /**
4  * 员工
5  */
6 public class ElementEmployeeImpl extends Element {
7
8     /**
9      * 构造方法
10     */
11     @param name
12     */
13     public ElementEmployeeImpl(String name) {
14         super(name);
15     }
16
17     /**
18      * 双重分派
19      */
20     @Override
21     public void accept(IVisitor visitor) {
22         visitor.visit(this);
23     }
24
25 }
```

访问者接口:

代码片段11 IVisitor.java

```
1 package cn.steven.pattern.demo.visitor.pattern;
2
3 /**
4  * 访问者接口
5  */
6 public interface IVisitor {
```

```

7
8 /**
9  * 访问领导的方法
10 */
11 void visit(ElementLeaderImpl element);
12
13 /**
14  * 访问员工的方法
15 */
16 void visit(ElementEmployeeImpl element);
17 }

```

问好的访问者:

代码片段12 VisitorHelloImpl.java

```

1 package cn.steven.pattern.demo.visitor.pattern;
2 /**
3  * 问好访问者
4  */
5 public class VisitorHelloImpl implements IVisitor {
6
7     @Override
8     public void visit(ElementLeaderImpl element) {
9         System.out.println(element.getName()+" 领导好 ★");
10    }
11
12    @Override
13    public void visit(ElementEmployeeImpl element) {
14        System.out.println(element.getName()+" 你好");
15    }
16
17 }

```

工作访问者:

代码片段13 VisitorWorkImpl.java

```

1 package cn.steven.pattern.demo.visitor.pattern;
2
3 /**
4  * 工作访问者
5  */
6 public class VisitorWorkImpl implements IVisitor {
7
8     @Override
9     public void visit(ElementLeaderImpl element) {
10        System.out.println("向 " + element.getName() + " 汇报工作 ★");
11    }
12
13    @Override
14    public void visit(ElementEmployeeImpl element) {
15        System.out.println("与 " + element.getName() + " 一起工作");
16    }
17 }

```


描述结构的类:

代码片段14 ObjectStructure.java

```

1 package cn.steven.pattern.demo.visitor.pattern;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Random;
6
7 /**
8  * 对象结构
9  */
10 public class ObjectStructure {
11
12     /**
13      * 人员列表
14      */
15     private List<Element> eList;
16
17     /**
18      * 用于随机生成人员
19      */
20     private static Random rand = new Random();
21
22     /**
23      * 随机人员数量
24      */
25     private int elementCount = 5;
26
27     public List<Element> getEList() {
28         return eList;
29     }
30
31     public void setEList(List<Element> list) {
32         eList = list;
33     }
34
35     /**
36      * 构造方法生成随机列表
37      */
38     public ObjectStructure() {
39         eList = new ArrayList<Element>();
40         for (int i = 0; i < elementCount; i++) {
41             int num = rand.nextInt(2);
42             switch (num) {
43                 case 0:
44                     eList.add(new ElementLeaderImpl("领导" + i));

```

```
45         break;
46     case 1:
47         eList.add(new ElementEmployeeImpl("员工" + i));
48         break;
49     default:
50         throw new RuntimeException("不支持的类索引！");
51     }
52 }
53 }
54
55 }
```

客户端代码:

代码片段15 Client.java

```
1 package cn.steven.pattern.demo.visitor.pattern;
2
3 /**
4  * 访问者模式
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 创建需要的变量
12          */
13         ObjectStructure struc = new ObjectStructure();
14         IVisitor helloVisitor = new VisitorHelloImpl();
15         IVisitor workVisitor = new VisitorWorkImpl();
16
17         /**
18          * 对集合对象问好
19          */
20         for (Element e : struc.getEList()) {
21             e.accept(helloVisitor);
22         }
23
24         /**
25          * 对集合对象工作
26          */
27         for (Element e : struc.getEList()) {
28             e.accept(workVisitor);
29         }
30     }
31 }
32
33 }
```

运行结果如图20-8所示。

由结果可见，目前迭代的Element元素已经具备了访问者提供的功能，由于双重分派的作用，现在也可以正确地调用特定的方法了。

20.2.3 用访问者模式设计灵活高效的收银程序

在本章第一节中，我们已经讨论过收银员的商品处理效率低下的问题。通过上一节的学习，我们已经初步掌握了访问者模式的设计和使用方法，在本节中，将会继续扩展其应用。下面我们的设计目标是不改变商品类，使用访问者为其增加功能，比如计费功能，此功能的目的是为不同的商品进行打折计费，比如玩具类打8折，食品类打7.5折等。在以上功能的基础上，还将涉及一个集合类，此类的功能是使用组合模式完成对一个对象组合的表示。

解决方案的设计类图如图20-9所示。

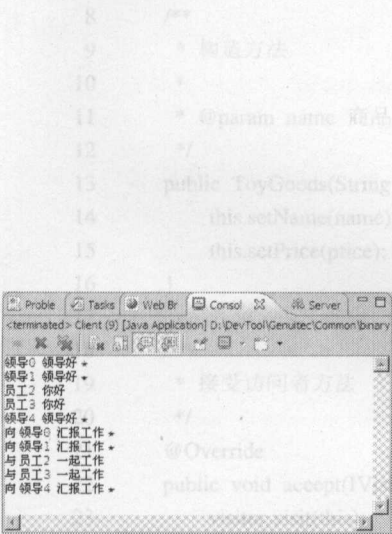


图20-8 运行结果

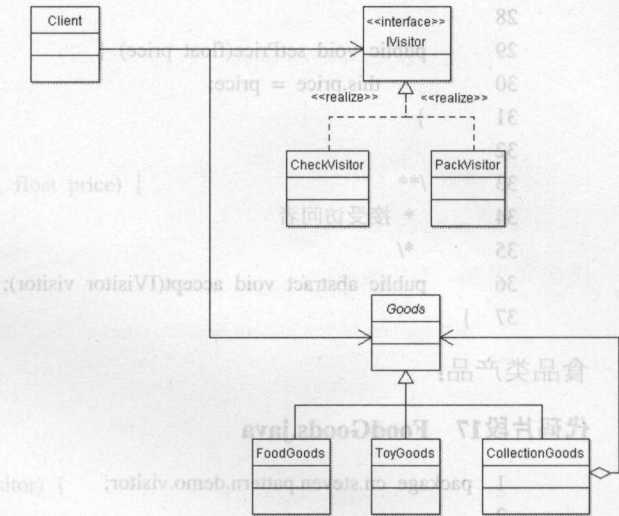


图20-9 设计类图

注意，图20-9中的CollectionGoods类聚合了其父类，这里使用的是组合模式的设计思想。下面首先看一下产品类的层次。

抽象产品类：

代码片段16 Goods.java

```
1 package cn.steven.pattern.demo.visitor;
2
3 /**
4  * 商品
5  */
6 public abstract class Goods {
7     /**
8      * 商品名
9      */
10    private String name;
11
12    /**
13     * 价格
```



```

14     */
15     private float price;
16
17     public String getName() {
18         return name;
19     }
20
21     public void setName(String name) {
22         this.name = name;
23     }
24
25     public float getPrice() {
26         return price;
27     }
28
29     public void setPrice(float price) {
30         this.price = price;
31     }
32
33     /**
34      * 接受访问者
35      */
36     public abstract void accept(IVisitor visitor);
37 }

```

食品类产品:

代码片段17 FoodGoods.java

```

1 package cn.steven.pattern.demo.visitor;
2
3 /**
4  * 食物商品
5  */
6 public class FoodGoods extends Goods {
7
8     /**
9      * 构造方法
10      *
11      * @param name
12      *      * 商品名
13      */
14     public FoodGoods(String name, float price) {
15         this.setName(name);
16         this.setPrice(price);
17     }
18
19     /**
20      * 接受访问者方法
21      */
22     @Override
23     public void accept(IVisitor visitor) {

```

```

24         visitor.visit(this);
25     }
26 }

```

玩具类产品:

代码片段18 ToyGoods.java

```

1  package cn.steven.pattern.demo.visitor;
2
3  /**
4   * 玩具商品
5   */
6  public class ToyGoods extends Goods {
7
8      /**
9       * 构造方法
10
11      * @param name 商品名
12      */
13     public ToyGoods(String name, float price) {
14         this.setName(name);
15         this.setPrice(price);
16     }
17
18     /**
19      * 接受访问者方法
20      */
21     @Override
22     public void accept(IVisitor visitor) {
23         visitor.visit(this);
24     }
25 }

```

组合类产品:

代码片段19 CollectionGoods.java

```

1  package cn.steven.pattern.demo.visitor;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5
6  /**
7   * 组合商品
8   */
9  public class CollectionGoods extends Goods {
10
11     private Collection<Goods> goodsCollection
12         = new ArrayList<Goods>();
13
14     public Collection<Goods> getGoodsCollection() {
15         return goodsCollection;
16     }
17 }

```

```

16     }
17
18     public void setGoodsCollection(
19         Collection<Goods> goodsCollection) {
20         this.goodsCollection = goodsCollection;
21     }
22
23     /**
24      * 组合对象接受访问者方法
25      */
26     @Override
27     public void accept(IVisitor visitor) {
28         for (Goods g : goodsCollection) {
29             g.accept(visitor);
30         }
31         visitor.visit(this);
32     }
33 }

```

注意将代码片段19中第23行~第32行的代码实现与代码片段18中的方法进行对比学习。
访问者接口：

代码片段20 IVisitor.java

```

1 package cn.steven.pattern.demo.visitor;
2
3 /**
4  * 访问者接口
5  */
6 public interface IVisitor {
7
8     void visit(FoodGoods g);
9
10    void visit(ToyGoods g);
11
12    void visit(CollectionGoods g);
13 }

```

包装功能的访问者：

代码片段21 PackVisitor.java

```

1 package cn.steven.pattern.demo.visitor;
2
3 /**
4  * 包装访问者
5  */
6 public class PackVisitor implements IVisitor {
7
8     @Override
9     public void visit(FoodGoods g) {
10         System.out.println("★ 食品包装: " + g.getName());
11     }

```



```
12
13 @Override
14 public void visit(ToyGoods g) {
15     System.out.println("☆ 玩具包装: " + g.getName());
16 }
17
18 @Override
19 public void visit(CollectionGoods g) {
20     System.out.println("组合对象分拆包装");
21 }
22
23 }
```

计算金额访问者:

代码片段22 CheckVisitor.java

```
1 package cn.steven.pattern.demo.visitor;
2
3 /**
4  * 结账访问者
5  */
6 public class CheckVisitor implements IVisitor {
7
8     /**
9      * 总金额
10     */
11     private double countMoney = 0;
12
13     public double getCountMoney() {
14         return countMoney;
15     }
16
17     public void setCountMoney(double countMoney) {
18         this.countMoney = countMoney;
19     }
20
21     @Override
22     public void visit(FoodGoods g) {
23         double d = g.getPrice() * 0.75;
24         countMoney += d;
25         System.out.println("★ 食品7.5折: \t" + d);
26     }
27
28     @Override
29     public void visit(ToyGoods g) {
30         double d = g.getPrice() * 0.8;
31         countMoney += d;
32         System.out.println("☆ 玩具8折: \t" + d);
33     }
34
35     @Override
```

```

36     public void visit(CollectionGoods g) {
37         System.out.println("组合对象分拆计算");
38     }
39
40 }

```

客户端代码:

代码片段23 Client.java

```

1  package cn.steven.pattern.demo.visitor;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5
6  /**
7   * 访问者模式
8   */
9  public class Client {
10
11     public static void main(String[] args) {
12
13         /**
14          * 创建访问者对象
15          */
16         PackVisitor v1 = new PackVisitor();
17         CheckVisitor v2 = new CheckVisitor();
18
19         /**
20          * 创建集合
21          */
22         Collection<Goods> gCollection = new ArrayList<Goods>();
23
24         /**
25          * 加入两个普通对象
26          */
27         Goods a = new FoodGoods("凉粉", 5.3f);
28         Goods b = new ToyGoods("魔方", 35.9f);
29         gCollection.add(a);
30         gCollection.add(b);
31
32         /**
33          * 加入组合对象, cga, 结构如下所示
34          * cgb = (a,b)
35          * cga = (a,b,cgb)
36          */
37
38         CollectionGoods cga = new CollectionGoods();
39
40         CollectionGoods cgb = new CollectionGoods();
41         cgb.getGoodsCollection().add(a);
42         cgb.getGoodsCollection().add(b);
43

```

```
44 cga.getGoodsCollection().add(a);
45 cga.getGoodsCollection().add(b);
46 cga.getGoodsCollection().add(cgb);
47
48 /**
49  * 访问
50  */
51 System.out.println("访问包装功能");
52 a.accept(v1);
53 b.accept(v1);
54 cga.accept(v1);
55
56 System.out.println("访问计费功能");
57 a.accept(v2);
58 b.accept(v2);
59 cga.accept(v2);
60 System.out.println("金额打折后合计: \t"+v2.getCountMoney());
61
62 }
63
64 }
```

运行结果如图20-10所示。

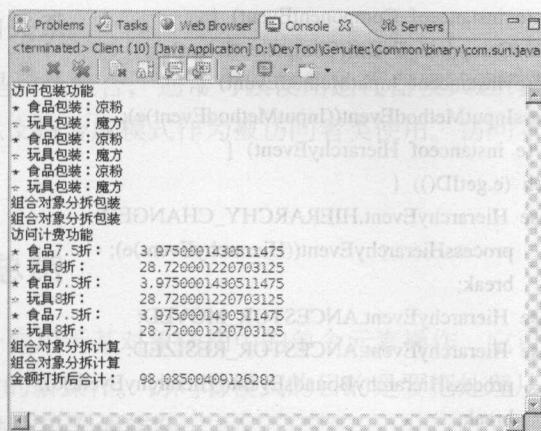


图20-10 运行结果

如上所示，使用了访问者模式的设计结构，现在终于可以在不更改元素类的情况下为元素增加功能了，而且可以和组合模式一起使用，使得功能更强大。

20.2.4 访问者模式在JDK中的实例

访问者模式自身有一个很严重的缺点，就是元素类型的改动会影响访问者模式的全体变化，所以在JDK中由于需要顾及到可扩展性，使用的情况很少。

但是JDK中也会遇到需要分派的问题，如下代码所示：

代码片段24 java.awt.Component.java 节选

```
1 protected void processEvent(AWTEvent e) {
2     if (e instanceof FocusEvent) {
```



```

3      processFocusEvent((FocusEvent)e);
4
5      } else if (e instanceof MouseEvent) {
6          switch(e.getID()) {
7              case MouseEvent.MOUSE_PRESSED:
8              case MouseEvent.MOUSE_RELEASED:
9              case MouseEvent.MOUSE_CLICKED:
10             case MouseEvent.MOUSE_ENTERED:
11             case MouseEvent.MOUSE_EXITED:
12                 processMouseEvent((MouseEvent)e);
13                 break;
14             case MouseEvent.MOUSE_MOVED:
15             case MouseEvent.MOUSE_DRAGGED:
16                 processMouseMotionEvent((MouseEvent)e);
17                 break;
18             case MouseEvent.MOUSE_WHEEL:
19                 processMouseWheelEvent((MouseWheelEvent)e);
20                 break;
21         }
22     } else if (e instanceof KeyEvent) {
23         processKeyEvent((KeyEvent)e);
24     } else if (e instanceof ComponentEvent) {
25         processComponentEvent((ComponentEvent)e);
26     } else if (e instanceof InputMethodEvent) {
27         processInputMethodEvent((InputMethodEvent)e);
28     } else if (e instanceof HierarchyEvent) {
29         switch (e.getID()) {
30             case HierarchyEvent.HIERARCHY_CHANGED:
31                 processHierarchyEvent((HierarchyEvent)e);
32                 break;
33             case HierarchyEvent.ANCESTOR_MOVED:
34             case HierarchyEvent.ANCESTOR_RESIZED:
35                 processHierarchyBoundsEvent((HierarchyEvent)e);
36                 break;
37         }
38     }
39 }
40 }
41 }

```

代码片段24与代码片段4使用的是相同的思想，因为这段代码中调用方法将会有Component（对象自身的类型）和Event（参数类型）两个需要在运行时刻才能确定其具体类型的变量，所以必须要进行再次的分派才能正常运行，因为Java只支持静态多分派，而不支持动态多分派。

20.2.5 访问者模式的使用范围及优点

访问者模式的优点：

- 访问者模式使得增加新的操作变得很容易。如果一些操作依赖于一个复杂的结构对象的话，那么一般而言，增加新的操作会很复杂。而使用访问者模式，增加新的操作就意味着增加一个新的访问者类，因此，会变得很容易。

- 访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。
- 访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。迭代类只能访问属于同一个类型等级结构的成员对象，而不能访问属于不同等级结构的对象。

- 积累状态。在访问者模式中，每一个单独的访问者对象都集中了相关的行为，这就可以在访问的过程中将执行操作的状态积累在自己内部，而不是分散到很多的节点对象中。这是其有益于系统维护的优点。

访问者模式的缺点：

- 增加新的节点类变得很困难。每增加一个新的节点都意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个具体访问者类中增加相应的具体操作。

- 破坏封装。访问者模式要求访问者对象访问并调用每一个节点对象的操作，这隐含了一个对所有节点对象提出的要求：它们必须暴露一些自己的操作和内部状态。不然，访问者的访问就变得没有意义。由于访问者对象自己会积累访问操作所需的状态，使这些状态不再存储在节点对象中，因此这也是破坏封装的。

使用访问者模式的情景：

访问者模式应该用在被访问类结构比较稳定的时候，换言之，就是系统很少出现增加新节点的情况，因为访问者模式对开一闭原则的支持并不好。

在一个被访问类的层次会剧烈变动的需求下，使用访问者模式是不合适的。

20.2.6 与其他模式的关系

如果被访问的对象是一个集合，通常可以使用迭代器模式进行辅助索引元素。

访问者模式通常可以支持合成模式作为被访问者类使用。访问者模式可以参与解释器模式的实现。

20.3 访问者模式总结

访问者模式表示一个作用于某对象结构中的单个元素操作。它可以在不改变各元素类的前提下定义作用于这些元素的新操作。访问者模式的目的是要把处理从数据结构中分离出来，从而对客户端的使用代码进行极大的简化。

访问者模式在Java实现中采用了双重分派的编程方法。

在类剧烈变动的情况下不宜使用访问者模式，读者可以采用其他模式进行需求的实现。

第21章 状态模式 (State)

状态模式允许一个对象在其内部状态改变时更改它的行为，这使得对象看起来就像是修改了它的类¹一样。

一个实体对待外部事物所表现出来的反应从一个角度反映了它当时的状态。在实际生活中有很多这样关于状态的例子，比如要求一个人帮忙完成一个私人的事情，如果那个人现在很忙，心情不好，那么现在去说这件事他通常是不会答应的，但是等个那人心情好了再说，结果可能就不一样了。

再比如公司的电话有可能被锁定了长途功能，锁定了之后按0键会播放忙音，没有锁定则可以正常打长途电话，按0键没有任何提示音。

以上例子就说明在实体处于不同的状态下，对于操作的方式和结果是不同的，如果需要通过编程来实现状态功能，通常在实现中会加入很多判断状态的语句如if、switch等，但是这样做的可扩展性并不好。本章介绍的状态模式就是用于解决类似和状态有关的问题的。

21.1 如何调整超市的运营状态

超市每日的运营状态为：

- 关门：任何人不允许进入。可转换为准备开门状态。
- 准备开门：员工可以进入，顾客不许进入。可转换为关门、开门状态。
- 开门：所有人可以进入。可转换为关门状态。

由此可见，不同的状态下进入超市的行为结果是不一样的，在不同的状态下，超市表现出来的行为大不一样，仿佛具有完全不同的类特性一样。

初步的实现代码如下，商店类：

代码片段1 Shop.java

```
1 package cn.steven.pattern.demo.state.quest;
2
3 /**
4  * 商店
5  */
6 public class Shop {
7     /**
8      * 商店状态 "ready" 准备开门, "open" 开门, "close" 关门
9      */
10    private String state;
11
12    /**
```

¹Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.GOF[95]


```
13     * 商店名称
14     */
15     private String name;
16
17     /**
18     * 构造方法
19     *
20     * @param state
21     *         商店状态
22     * @param name
23     *         商店名称
24     */
25     public Shop(String name, String state) {
26         super();
27         this.state = state;
28         this.name = name;
29     }
30
31     /**
32     * 商店 "准备开门" 方法
33     */
34     public void ready() {
35         /**
36          * 只有当状态是close时准备开门才是合适的
37          */
38         if (state.equals("close")) {
39             state = "ready";
40         } else {
41             throw new RuntimeException("状态不正确");
42         }
43     }
44
45     /**
46     * 商店 "开门" 方法
47     */
48     public void open() {
49         /**
50          * 只有当状态是ready时开门才是合适的
51          */
52         if (state.equals("ready")) {
53             state = "open";
54         } else {
55             throw new RuntimeException("状态不正确");
56         }
57     }
58
59     /**
60     * 商店 "关门"方法
61     */
62     public void close() {
63         /**
```

```

64         * 只有当状态是ready或open时关门才是合适的
65         */
66         if (state.equals("ready") || state.equals("open")) {
67             state = "close";
68         } else {
69             throw new RuntimeException("状态不正确");
70         }
71     }
72
73     /**
74      * 查询状态
75      */
76     public void checkState() {
77         if (state.equals("open")) {
78             System.out.println(name + "开门啦！");
79         }
80         if (state.equals("ready")) {
81             System.out.println(name + "准备开门啦！");
82         }
83         if (state.equals("close")) {
84             System.out.println(name + "关门了！");
85         }
86     }
87
88 }

```

客户端使用的代码如下：

代码片段2 Client.java

```

1 package cn.steven.pattern.demo.state.quest;
2
3 /**
4  * 商店状态
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9         Shop shop = new Shop("总店", "close");
10        shop.checkState();
11        shop.ready();
12        shop.checkState();
13        shop.open();
14        shop.checkState();
15        shop.close();
16        shop.checkState();
17
18        /**
19         * 以下代码违反了约定，所以出错了
20         */
21        shop.open();
22        shop.checkState();

```

```
23     }  
24  
25 }
```

执行结果如图21-1所示。

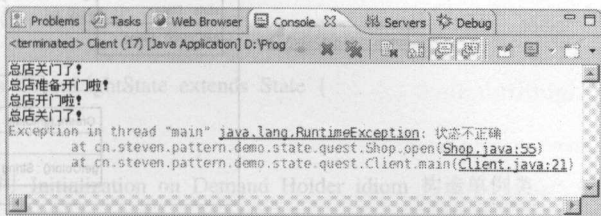


图21-1 Client.java执行结果

由此可见，功能虽然已经实现，但是代码中充斥着大量的判断语句，使得编程变得十分复杂，随着状态的增多，判断的代码量将会显著增长。

出现这个问题的原因是由于含有状态的实体中除了包括维护其自身的信息，还包含了有关状态的各种方法，由于很多行为和状态相关，所以代码就会很烦琐。

如果改用状态模式则可以将有关状态的代码隔离出来，使结构更加简洁合理。

21.2 状态模式的结构

状态模式的结构要点是将状态抽象为单独的类层次，用环境类聚合状态类，当调用与状态相关的方法时，环境对象会将其委托给状态对象。

21.2.1 状态模式

将状态分离成一个独立的类层次是状态模式的特点，而且这些状态和外部持有它的对象之间有很多联系，比较关键的一种操作是状态的切换。状态可以由外部环境对象进行切换，也可以由状态对象切换，两种方式需要根据不同的需求使用。很多情况下，状态对象会改变环境对象的属性，这时状态对象中需要被传入一个环境对象的参数。

下面使用一个生活中常见的交通信号灯的例子来说明状态模式。交通信号灯虽然有很多，但是状态却只有三种：红、绿、黄。这些状态其实是可以复用的，所以可以将灯状态设置为单例模式类。

每一种状态会具有不同的状态属性和方法，使用这个状态的外部对象需要将与状态相关的方法委托给特定状态对象。

交通信号灯演示图片如图21-2所示。其状态模式设计类图如图21-3所示。

图中的参与者如下：

- **Signal**: 上下文类，这个实例可以定义当前包含的状态。
- **State类**: 抽象状态类，定义一个接口以封装和定义一个与特定状态相关的行为。
- **xxxState类**: 具体状态类，每一个子类实现一个与具体状态相关的行为。

具体代码如下，状态抽象类：

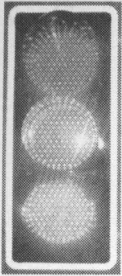


图21-2 三色交通信号灯

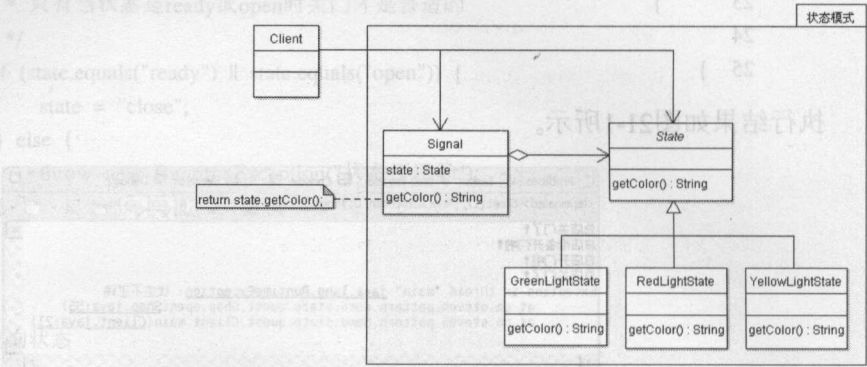


图21-3 状态模式类图

代码片段3 State.java

```
1 package cn.steven.pattern.demo.state.pattern;
2
3 /**
4  * 信号灯状态抽象类
5  */
6 abstract public class State {
7     /**
8      * 信号灯颜色
9      */
10    private String color;
11
12    public String getColor() {
13        return color;
14    }
15
16    public void setColor(String color) {
17        this.color = color;
18    }
19
20    /**
21     * 信号灯提示语音
22     */
23    private String msg;
24
25    public String getMsg() {
26        return msg;
27    }
28
29    public void setMsg(String msg) {
30        this.msg = msg;
31    }
32 }
```

绿灯状态:

代码片段4 GreenLightState.java

```

1 package cn.steven.pattern.demo.state.pattern;
2
3 /**
4  * 绿灯状态, 单例模式
5  */
6 public class GreenLightState extends State {
7
8     /**
9      * 采用 Initialization on Demand Holder idiom 构造单例类
10     *
11     * 内部静态类, 单例持有者
12     */
13     static class SingletonHolder {
14         static GreenLightState instance = new GreenLightState();
15     }
16
17     public static GreenLightState getInstance() {
18         return SingletonHolder.instance;
19     }
20
21     private GreenLightState() {
22         this.setColor("绿色");
23         this.setMsg("可以通行。");
24     }
25 }

```

红灯状态:

代码片段5 RedLightState.java

```

1 package cn.steven.pattern.demo.state.pattern;
2
3 /**
4  * 红灯状态, 单例模式
5  */
6 public class RedLightState extends State {
7
8     /**
9      * 采用 Initialization on Demand Holder idiom 构造单例类
10     *
11     * 内部静态类, 单例持有者
12     */
13     static class SingletonHolder {
14         static RedLightState instance = new RedLightState();
15     }
16
17     public static RedLightState getInstance() {
18         return SingletonHolder.instance;
19     }
20

```

```

21     private RedLightState() {
22         this.setColor("红色");
23         this.setMsg("禁止通行！");
24     }
25 }

```

黄灯状态:

代码片段6 YellowLightState.java

```

1  package cn.steven.pattern.demo.state.pattern;
2
3  /**
4   * 黄灯状态，单例模式
5   */
6  public class YellowLightState extends State {
7
8      /**
9       * 采用 Initialization on Demand Holder idiom 构造单例类
10      *
11      * 内部静态类，单例持有者
12      */
13      static class SingletonHolder {
14          static YellowLightState instance = new YellowLightState();
15      }
16
17      public static YellowLightState getInstance() {
18          return SingletonHolder.instance;
19      }
20
21      private YellowLightState() {
22          this.setColor("黄色");
23          this.setMsg("注意马上要改变状态，目前可以保持刚才的通行状态。");
24      }
25 }

```

注意状态的具体实现类通常是全局共享的，所以实现为单例类，实现的具体方法使用的是懒加载的方式：Initialization on Demand Holder idiom¹。

上下文类：

代码片段7 Signal.java

```

1  package cn.steven.pattern.demo.state.pattern;
2
3  /**
4   * 具有三种状态的交通信号灯
5   */
6  public class Signal {
7
8      /**

```

¹http://en.wikipedia.org/wiki/Initialization_on_demand_holder_idiom.


```
9      * 聚合状态对象
10     */
11     private State state;
12
13     /**
14      * 设置状态
15      */
16     public void setState(State state) {
17         this.state = state;
18
19         /**
20          * 通知
21          */
22         System.out.println("注意, 目前信号灯的颜色是: " + this.getColor());
23         System.out.println(this.getMsg());
24     }
25
26     /**
27      * 得到当前灯颜色
28      */
29     public String getColor() {
30         return state.getColor();
31     }
32
33     /**
34      * 得到提示文字
35      */
36     public String getMsg() {
37         return state.getMsg();
38     }
39
40 }
```

客户端代码:

代码片段8 Client.java

```
1 package cn.steven.pattern.demo.state.pattern;
2
3 /**
4  * 状态模式
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 构建上下文对象
12          */
13         Signal signal = new Signal();
14
15         /**
16          * 改变状态
```

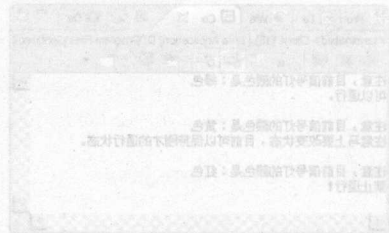
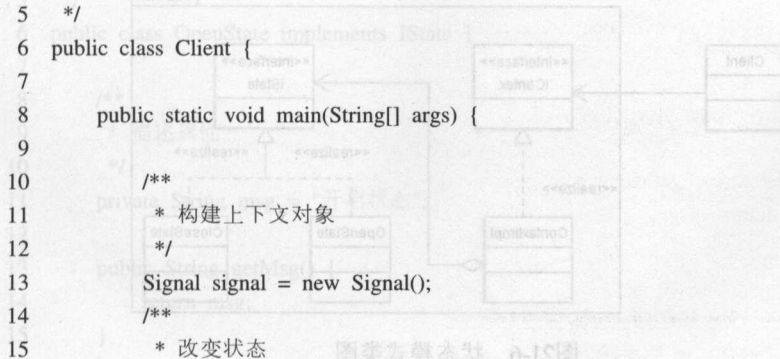


图 21-4 运行结果图



```
16      */
17      signal.setState(GreenLightState.getInstance());
18      System.out.println();
19      signal.setState(YellowLightState.getInstance());
20      System.out.println();
21      signal.setState(RedLightState.getInstance());
22
23  }
24
25 }
```

代码运行结果如图21-4所示。

由以上实例可见，通过状态模式可以有效地封装状态的各种变化因素，使外部环境类的编码得到极大的简化。状态模式的实现类通常被设计为单例的，这样可以使各种需要使用状态的环境对象共用状态对象，但需要注意多线程情况下的临界资源问题。

如果状态类不设计为单例模式类，也可以令环境类在使用的时候实例化状态对象，在切换状态时丢弃上一个状态。

在状态模式中还有一个问题就是关于状态切换的方法问题。上面的例子中是使用环境对象的方法来设置状态，通常情况下这样做是很好的，但是有时外界也需要知道状态的规则，所以就增加了复杂度。根据迪米特法则（LoD）-最少知识原则，我们可以将状态的切换方法放置于状态对象中。

试想在生活中有一种电源开关，有打开和关闭两种状态，但是却只有一种操作方法，就是“触摸”，如图21-5所示。

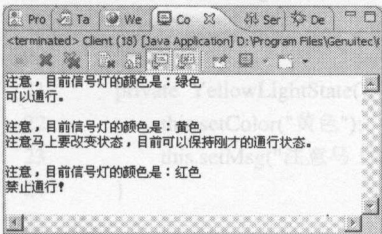


图21-4 状态模式运行结果

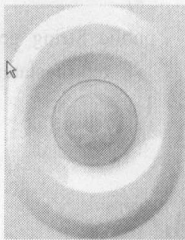


图21-5 触摸式电源开关

使用这种开关，无论当前开关的状态如何，只要触摸后开关状态就会自动切换至相反的状态。在设计其软件结构时，需要用状态对象控制外部环境，所以设计的类图如图21-6所示。

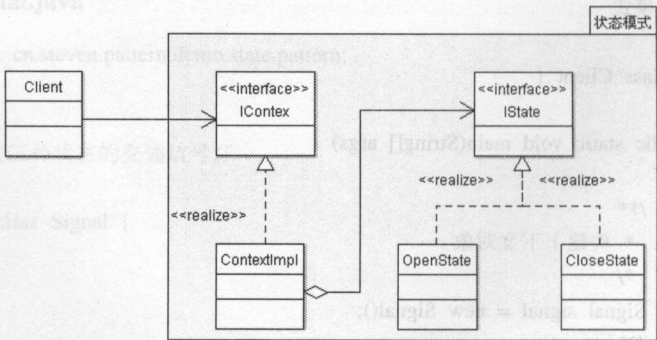


图21-6 状态模式类图

注意图21-6和图21-3的区别在于前者多增加了一个上下文的接口，这个接口的作用是为了让状态可以访问上下文类。

两个状态类的关系如图21-7所示。

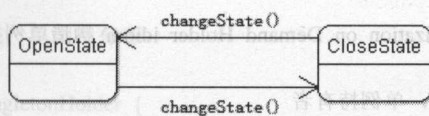


图21-7 状态图

状态接口：

代码片段9 IState.java

```
1 package cn.steven.pattern.demo.state.pattern2;
2
3 /**
4  * 状态接口
5  */
6 public interface IState {
7
8     /**
9      * 更改状态
10     */
11     void changeState(IContext context);
12
13     /**
14      * 获得描述
15      * @return
16      */
17     String getMsg();
18 }
```

开启状态：

代码片段10 OpenState.java

```
1 package cn.steven.pattern.demo.state.pattern2;
2
3 /**
4  * 开启状态
5  */
6 public class OpenState implements IState {
7
8     /**
9      * 描述属性
10     */
11     private String msg = "开启状态";
12
13     public String getMsg() {
14         return msg;
15     }
16 }
```



```
17 public void setMsg(String msg) {
18     this.msg = msg;
19 }
20
21 /**
22  * 采用 Initialization on Demand Holder idiom 构造单例类
23  *
24  * 内部静态类，单例持有者
25  */
26 static class SingletonHolder {
27     static OpenState instance = new OpenState();
28 }
29
30 public static OpenState getInstance() {
31     return SingletonHolder.instance;
32 }
33
34 private OpenState() {
35 }
36
37 /**
38  * 切换状态
39  */
40 @Override
41 public void changeState(IContext context) {
42     context.setState(CloseState.getInstance());
43 }
44 }
45 }
```

关闭状态:

代码片段11 CloseState.java

```
1 package cn.steven.pattern.demo.state.pattern2;
2
3 /**
4  * 关闭状态
5  */
6 public class CloseState implements IState {
7
8     /**
9      * 描述属性
10     */
11     private String msg = "关闭状态";
12
13     public String getMsg() {
14         return msg;
15     }
16
17     public void setMsg(String msg) {
18         this.msg = msg;
```

```

19 }
20
21 /**
22  * 采用 Initialization on Demand Holder idiom 构造单例类
23  *
24  * 内部静态类，单例持有者
25  */
26 public static class SingletonHolder {
27     static CloseState instance = new CloseState();
28 }
29
30 public static CloseState getInstance() {
31     return SingletonHolder.instance;
32 }
33
34 private CloseState() {
35
36 }
37
38 /**
39  * 切换状态，转换为OpenState
40  */
41 @Override
42 public void changeState(IContext context) {
43     context.setState(OpenState.getInstance());
44 }
45 }

```

上下文接口:

代码片段12 IContext.java

```

1 package cn.steven.pattern.demo.state.pattern2;
2
3 /**
4  * 上下文接口
5  */
6 public interface IContext {
7
8     /**
9      * 设置状态方法
10      *
11      * @param state
12      */
13     void setState(IState state);
14
15     /**
16      * 更改状态方法
17      */
18     void changeState();
19
20     /**

```

```
21      * 调用状态数据
22      */
23      public void showMsg();
24  }
```

上下文实现类:

代码片段13 ContextImpl.java

```
1  package cn.steven.pattern.demo.state.pattern2;
2
3  public class ContextImpl implements IContext {
4
5      /**
6       * 状态属性
7       */
8      private IState state;
9
10     public IState getState() {
11         return state;
12     }
13
14     public void setState(IState state) {
15         this.state = state;
16     }
17
18     /**
19      * 构造方法
20      */
21     public ContextImpl() {
22         /**
23          * 初始化时给定一个状态
24          */
25         this.setState(CloseState.getInstance());
26     }
27
28     /**
29      * 切换状态方法
30      */
31     public void changeState() {
32         state.changeState(this);
33     }
34
35     /**
36      * 调用状态数据
37      */
38     public void showMsg() {
39         System.out.println(state.getMsg());
40     }
41
42 }
```

客户端代码:

代码片段14 Client.java

```
1 package cn.steven.pattern.demo.state.pattern2;
2
3 /**
4  * 状态模式
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 创建环境
12          */
13         IContext context = new ContextImpl();
14         context.showMsg();
15
16         /**
17          * 更改状态
18          */
19         context.changeState();
20         context.showMsg();
21
22         context.changeState();
23         context.showMsg();
24
25         context.changeState();
26         context.showMsg();
27     }
28 }
29
30 }
```

代码运行结果图21-8所示。

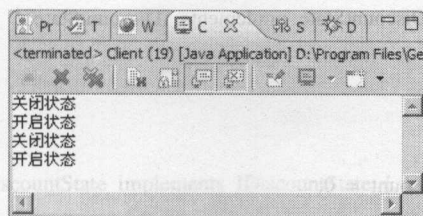


图21-8 运行结果截图

参照代码片段14和图21-8可见，虽然客户端代码并没有显式地给出需要切换到状态，但是系统却能自动切换，非常方便。注意这种状态模式的实现需要状态间的切换规则十分明确才可以。

21.2.2 用状态模式设计超市在不同时段的打折状态

超市常常会在不同的时间段设计不同的折扣幅度，用以吸引客户购买，但是折扣的状态多种多样，所以需要使用状态模式来为其设计架构。下面就是给出的示例代码，由于设计类图与

图21-3相似，所以这里没有重复给出。

折扣状态接口：

代码片段15 IDiscountState.java

```
1 package cn.steven.pattern.demo.state;
2
3 /**
4  * 折扣状态接口
5  */
6 public interface IDiscountState {
7     /**
8      * 折扣幅度 0~1
9      *
10     * @return
11     */
12     float getDiscount();
13
14     /**
15      * 文字描述
16      *
17     * @return
18     */
19     String getMsg();
20 }
```

全价状态：

代码片段16 NormalDiscountState.java

```
1 package cn.steven.pattern.demo.state;
2
3 /**
4  * 不参加折扣
5  */
6 public class NormalDiscountState implements IDiscountState {
7
8     /**
9      * 折扣
10     */
11     private float discount = 0;
12
13     public float getDiscount() {
14         return discount;
15     }
16
17     public void setDiscount(float discount) {
18         this.discount = discount;
19     }
20
21     /**
22      * 状态描述
23      */
24 }
```

```

24 private String msg = "没有折扣";
25
26 public String getMsg() {
27     return msg;
28 }
29
30 public void setMsg(String msg) {
31     this.msg = msg;
32 }
33
34 /**
35  * 采用 Initialization on Demand Holder idiom 构造单例类
36  * System.out.println(discountState.getMsg());
37  * 内部静态类，单例持有者
38  */
39 static class SingletonHolder {
40     static NormalDiscountState instance
41         = new NormalDiscountState();
42 }
43
44 public static NormalDiscountState getInstance() {
45     return SingletonHolder.instance;
46 }
47
48 private NormalDiscountState() {
49 }
50
51
52 }

```

半价状态:

代码片段17 HalfDiscountState.java

```

1 package cn.steven.pattern.demo.state;
2
3
4 /**
5  * 半价
6  */
7 public class HalfDiscountState implements IDiscountState {
8
9     /** Shop shop = new Shop();
10      * 折扣
11      */
12     private float discount = 0.5f;
13
14     public float getDiscount() {
15         return discount;
16     }
17
18     public void setDiscount(float discount) {
19         this.discount = discount;
20     }
21 }

```



```

20     }
21
22     /**
23      * 状态描述
24      */
25     private String msg = "半价";
26
27     public String getMsg() {
28         return msg;
29     }
30
31     public void setMsg(String msg) {
32         this.msg = msg;
33     }
34
35     /**
36      * 采用 Initialization on Demand Holder idiom 构造单例类
37      *
38      * 内部静态类，单例持有者
39      */
40     static class SingletonHolder {
41         static HalfDiscountState instance = new HalfDiscountState();
42     }
43
44     public static HalfDiscountState getInstance() {
45         return SingletonHolder.instance;
46     }
47
48     private HalfDiscountState() {
49
50     }
51
52 }

```

```

15 }
16
17 public void setDiscountState(IDiscountState discountState) {
18     this.discountState = discountState;
19 }
20
21 /**
22  * 结账
23  *
24  * @param money
25  */
26 public void check(float money) {
27     System.out.println(discountState.getMsg());
28     System.out.println("原价: " + money + "\t 现价"
29         + (money * (1 - discountState.getDiscount())));
30 }
31
32 /**
33  * 初始化方法
34  */
35 public Shop() {
36     this.setDiscountState(NormalDiscountState.getInstance());
37 }
38
39 }

```

客户端代码:

代码片段19 Client.java

```

1 package cn.steven.pattern.demo.state;
2
3 /**
4  * 客户端
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 构建环境对象
12          */
13         Shop shop = new Shop();
14         shop.check(100);
15
16         shop.setDiscountState(HalfDiscountState.getInstance());
17         shop.check(100);
18     }
19
20 }

```

代码的运行结果图21-9所示。



图21-9 折扣例子运行结果

由图21-9可见，由于引入了状态模式，现在超市已经可以通过设置一个状态对象来控制所有的商品价格的计算了，并且即使要增加一个折扣规则也是很简单的。在这里给出的是一个建议性的示例，如果情况复杂，并且需要更好的扩展性，建议考虑以下方案：

- 将折扣的类配置在外部XML文件中。

- 按照时间来控制折扣的设置，建议使用一个线程来控制状态的切换。
- 可以设计一个状态管理工具类统一管理各种状态。

21.2.3 状态模式的使用范围及优点

状态模式的优点：

- 封装转换过程，也就是转换规则，使规则扩展得非常简单。
- 环境对象聚合了状态对象，可以将操作委托给后者，从而使得编码非常简单。

状态模式的缺点：

- 状态类需要单独进行编码，整体结构比较复杂。

使用范围：

- 一个对象的行为取决于它的状态，并且必须在运行时刻根据状态改变它的行为的情况。
- 一个操作中含有庞大的多分支的条件语句，并且这些分支依赖于该对象的状态，这个状态通常用一个或多个枚举常量表示时。通常，有多个操作包含这一相同的条件结构时，状态模式会将每一个条件分支放入一个单独的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

21.2.4 与其他模式的关系

在大多数情况下状态模式中可以共享使用状态对象，所以可以用单例模式来构建状态。

策略模式在结构上与状态模式非常相似，但是在概念上，它们的目的差异非常大。区分这两个模式的关键是看行为是由状态驱动还是由一组算法驱动，这条规则似乎有点随意，但是在进行判断时还是需要考虑它。通常，状态模式的“状态”是在对象内部的，策略模式的“策略”可以在对象外部，不过这也不是一条严格、可靠的规则。

21.3 状态模式总结

状态模式主要用于当控制一个对象状态切换的条件表达式过于复杂时的情况。它通过把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简单化。

当一个对象行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。

读者应注意区分状态模式和策略模式，做到灵活应用。在使用过程中，可以对标准的状态模式稍做改动以使其更适合应用场景。

第22章 备忘录模式 (Memento)

备忘录模式可以在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后在需要时就可以将该对象恢复到原先保存的状态¹。

俗话说，世上难买后悔药，所以凡事讲究个“三思而后行”，但生活常常见到有人作“痛心疾首”状：当初我要是……。如果真的有电影《大话西游》中能令时光倒流的“月光宝盒”，那么这世上也许会少一些伤感与后悔了——当然这只能是痴人说梦。

但是，在我们的程序世界里，却有后悔药可买。今天我们要讲的备忘录模式便是程序世界里的“月光宝盒”。

22.1 服务器硬盘坏了，营业数据全丢了

超市内的工作非常繁忙，每一个顾客都可能对超市的营业数据产生影响，最典型的影响就是营业收入。超市的这些数据通常保存在一个服务器中，但是有一天，很不幸，雷击导致电路故障造成了停电。重启服务器之后工作人员发现系统无法启动了，维修人员检查之后说：“服务器的硬盘坏了！”。

电子产品的确不是完全可靠的，即使在银行这种事情也是可能发生的，但是通常来说银行的工作人员并不担心某块硬盘坏掉，因为他们已经做了非常完备的数据备份工作，其在硬盘层面上的备份方案通常使用RAID² (Redundant Array of Independent Disks, 独立磁盘冗余数组) 方案。其基本思想就是把多个相对便宜的磁盘组合起来，令其成为一个磁盘数组，使它们的性能达到甚至超过一个价格昂贵、容量巨大的硬盘。根据选择的版本不同，RAID比单个硬盘有以下一个或多个方面的好处：增强数据集成度，增强容错功能，增加处理量或容量。另外，磁盘数组对于计算机来说，就像是一个单独的硬盘或逻辑存储单元。

这种方式是对于数据备份来说非常行之有效的，但是如果不幸遇到人力不可挽回的重大事情，恐怕这种手段也是不够的。这时可以利用分布式存储和分布式备份来解决区域性的安全问题。

常用的备份方案是复制，下面展示的代码就是利用复制的方式对可能出现安全问题的对象加以保存，并可以在需要的时候加以恢复。

商店类：

代码片段1 Shop.java

```
1 package cn.steven.pattern.demo.memento.quest;
2
3 /**
4  * 具有保存状态的商店类
```

¹Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.GOF[95]

²<http://zh.wikipedia.org/wiki/RAID>。

```
5  */
6  public class Shop {
7
8      /**
9       * 营业收入
10      */
11     private float money;
12
13     public float getMoney() {
14         return money;
15     }
16
17     /**
18      * 当要设置新金额时，保存上一个金额
19      *
20      * @param money
21      */
22     public void setMoney(float money) {
23         save();
24         this.money = money;
25     }
26
27     /**
28      * 保存前一个收入金额
29      */
30     private float backupMoney;
31
32     /**
33      * 保存方法
34      */
35     public void save() {
36         backupMoney = money;
37         System.out.println(backupMoney + " 被保存");
38     }
39
40     /**
41      * 恢复方法
42      */
43     public void restore() {
44         money = backupMoney;
45         System.out.println(backupMoney + " 被恢复");
46     }
47
48     /**
49      * 展示当前金额
50      */
51     public void show() {
52         System.out.println("当前金额: " + money);
53     }
54 }
```

客户端代码:

代码片段2 Client.java

```
1 package cn.steven.pattern.demo.memento.quest;
2
3 /**
4  * 测试
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9         Shop shop = new Shop();
10        shop.show();
11        shop.setMoney(222);
12        shop.show();
13        shop.setMoney(1000);
14        shop.show();
15        shop.restore();
16        shop.show();
17    }
18 }
19
20 }
```

运行结果如图22-1所示。

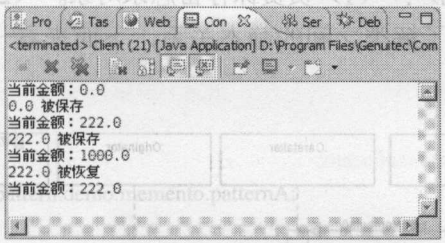


图22-1 执行结果

由图22-1可见，程序在每次修改金额前都会自动保存上一次的金额状态，但是仔细观察也会发现还存在一定的问题：

- 所保存的数据次数被限制为一次，可扩展性不好。
- 保存对象的功能和对象本身紧密耦合，无法复用。
- 没有恢复特定状态的功能。

以上缺陷将会在备忘录模式中得以解决。

22.2 备忘录模式的结构

备忘录模式可以将一个对象的状态外部化，并可以根据需要在适当的时候恢复状态。备忘录模式有两种实现方式：一种是白箱实现，一种是黑箱实现。它们各有优点，下面首先介绍一下结构较为简单的白箱实现方式。

22.2.1 备忘录模式白箱实现

白箱实现方式较为简单，其类图如图22-2所示。

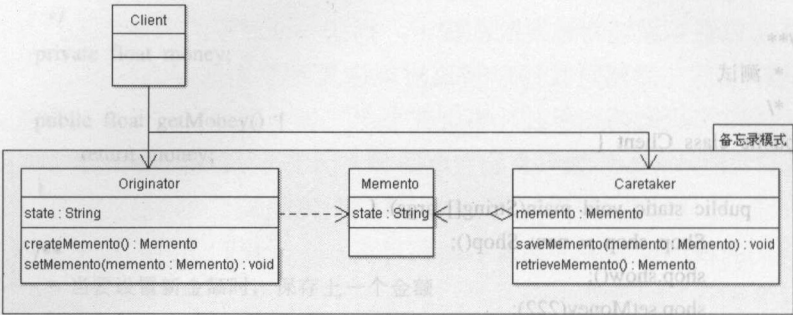


图22-2 备忘录模式类图（白箱）

图中参与者如下：

- **Memento**（备忘录角色）：（1）将原发器对象的内部状态存储起来。备忘录可以根据原发器对象的判断来决定存储多少原发器对象的内部状态。（2）备忘录可以保护其内容不被原发器对象之外的任何对象所读取。
- **Originator**（原发器角色）：（1）创建一个含有当前内部状态的备忘录对象。（2）使用备忘录对象存储其内部状态。
- **Caretaker**（负责人角色）：（1）负责保存备忘录对象。（2）不检查备忘录对象的内容。

如果只看类图的话只能了解类之间的静态关系，下面来看一下序列图以加深理解，如图22-3所示。

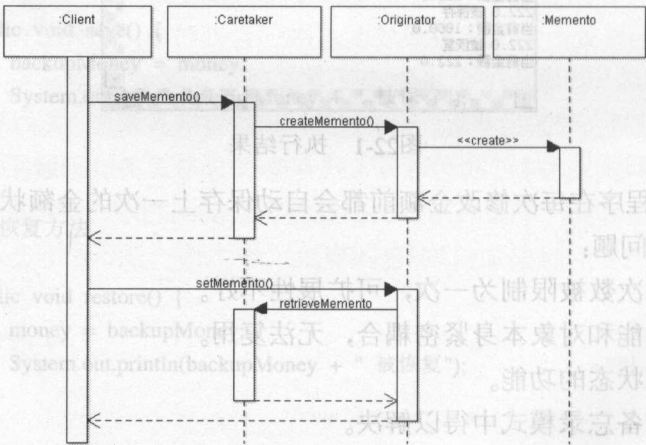


图22-3 备忘录模式序列图（白箱）

下面先展示备忘录类：

代码片段3 Memento.java

```
1 package cn.steven.pattern.demo.memento.patternA;
2
3 /**
```

```
4  /* 备忘录
5  */
6  public class Memento {
7      public class Client {
8          /**
9           * 保存的状态
10          */
11          private String state;
12          /**
13           * 创建备忘录
14           * 构造方法
15           * @param state
16           */
17          public Memento(String state) {
18              super();
19              this.state = state;
20          }
21          /**
22           * 构造方法
23           * @param state
24           */
25          public Memento(String state) {
26              super();
27              this.state = state;
28          }
29          /**
30           * 构造方法
31           * @param state
32           */
33          public Memento(String state) {
34              super();
35              this.state = state;
36          }
37          /**
38           * 构造方法
39           * @param state
40           */
41          public Memento(String state) {
42              super();
43              this.state = state;
44          }
45          /**
46           * 构造方法
47           * @param state
48           */
49          public Memento(String state) {
50              super();
51              this.state = state;
52          }
53          /**
54           * 构造方法
55           * @param state
56           */
57          public Memento(String state) {
58              super();
59              this.state = state;
60          }
61          /**
62           * 构造方法
63           * @param state
64           */
65          public Memento(String state) {
66              super();
67              this.state = state;
68          }
69          /**
70           * 构造方法
71           * @param state
72           */
73          public Memento(String state) {
74              super();
75              this.state = state;
76          }
77          /**
78           * 构造方法
79           * @param state
80           */
81          public Memento(String state) {
82              super();
83              this.state = state;
84          }
85          /**
86           * 构造方法
87           * @param state
88           */
89          public Memento(String state) {
90              super();
91              this.state = state;
92          }
93          /**
94           * 构造方法
95           * @param state
96           */
97          public Memento(String state) {
98              super();
99              this.state = state;
100             }
101         }
102     }
103 }
```

原发器类:

代码片段4 Originator.java

```
1  package cn.steven.pattern.demo.memento.patternA;
2
3  /**
4   * 原发器
5   */
6  public class Originator {
7
8      /**
9       * 状态
10      */
11      private String state;
12
13      /**
14       * 状态
15       */
16      public String getState() {
17          return state;
18      }
19
20      /**
21       * 状态
22       */
23      public void setState(String state) {
24          this.state = state;
25      }
26
27      /**
28       * 状态
29       */
30      public void setState(String state) {
31          this.state = state;
32      }
33
34      /**
35       * 状态
36       */
37      public void setState(String state) {
38          this.state = state;
39      }
40
41      /**
42       * 状态
43       */
44      public void setState(String state) {
45          this.state = state;
46      }
47
48      /**
49       * 状态
50       */
51      public void setState(String state) {
52          this.state = state;
53      }
54
55      /**
56       * 状态
57       */
58      public void setState(String state) {
59          this.state = state;
60      }
61
62      /**
63       * 状态
64       */
65      public void setState(String state) {
66          this.state = state;
67      }
68
69      /**
70       * 状态
71       */
72      public void setState(String state) {
73          this.state = state;
74      }
75
76      /**
77       * 状态
78       */
79      public void setState(String state) {
80          this.state = state;
81      }
82
83      /**
84       * 状态
85       */
86      public void setState(String state) {
87          this.state = state;
88      }
89
90      /**
91       * 状态
92       */
93      public void setState(String state) {
94          this.state = state;
95      }
96
97      /**
98       * 状态
99       */
100     public void setState(String state) {
101         this.state = state;
102     }
103 }
```

22.2.2 备忘录模式黑箱实现

黑箱实现可以理解为对原发器类开放宽接口而对备忘录类提供窄接口的实现。在实现之前，先来看一下宽接口的问题。在Java中支持一个类实现多个接口，如图22-5所示。

```
22      * 创建备忘录对象
23      *
24      * @return
25      */
26      public Memento createMemento() {
27          return new Memento(this.state);
28      }
29
30      /**
31       * 恢复状态
32       */
33      public void setMemento(Memento memento) {
34          this.state = memento.getState();
35      }
36
37  }
```

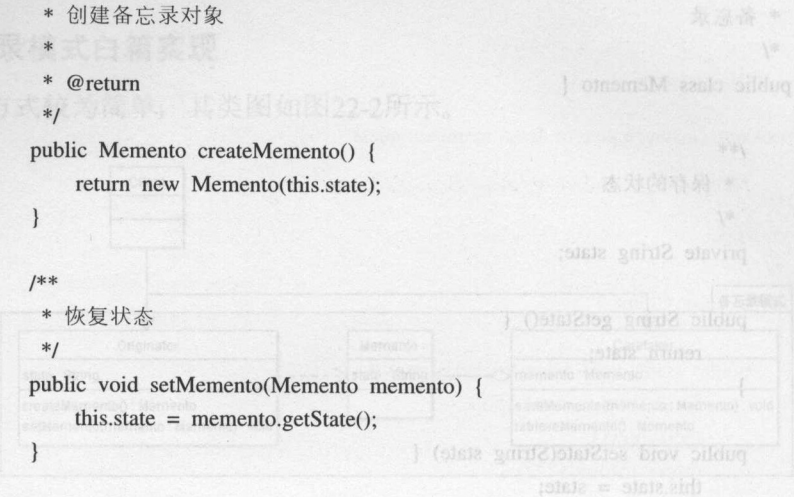


图22-3 备忘录模式类图（白箱）

负责人类：

代码片段5 Caretaker.java

```
1 package cn.steven.pattern.demo.memento.patternA;
2
3 /**
4  * 负责人
5  */
6 public class Caretaker {
7
8     /**
9      * 备忘录属性
10     */
11     private Memento memento;
12
13     /**
14      * 保存备忘录对象
15     */
16     public void saveMemento(Memento memento) {
17         this.memento = memento;
18     }
19
20     /**
21      * 取回备忘录对象
22     */
23     public Memento retrieveMemento() {
24         return memento;
25     }
26 }
```

客户端代码：

代码片段6 Client.java

```
1 package cn.steven.pattern.demo.memento.patternA;
2
```



```

3  /**
4   * 白箱备忘录模式
5   */
6   public class Client {
7
8       public static void main(String[] args) {
9
10          /**
11           * 创建源发起器
12           */
13          Originator originator = new Originator();
14
15          /**
16           * 创建负责人
17           */
18          Caretaker caretaker = new Caretaker();
19
20          /**
21           * 设置状态
22           */
23          originator.setState("初始状态");
24
25          /**
26           * 测试备忘录操作
27           */
28          System.out.println("当前状态: " + originator.getState());
29          System.out.println("保存备忘录");
30          caretaker.saveMemento(originator.createMemento());
31          System.out.println("设置新状态");
32          originator.setState("新状态");
33          System.out.println("当前状态: " + originator.getState());
34          System.out.println("恢复原状态");
35          originator.setMemento(caretaker.retrieveMemento());
36          System.out.println("当前状态: " + originator.getState());
37      }
38
39  }

```

代码的运行结果如图22-4所示。

由图22-4可见，现在已经可以成功地实现备忘录模式的功能了，但是这里还有一个比较严重的问题就是：白箱备忘录模式中使用的备忘录类可以被任意对象访问，这违反了模式中的定义。**Memento**类应该向**Originator**对象开放功能是对的，因为如果不这样的话就无法保存原发起器对象的状态，但是**Memento**类不应该向**Caretaker**对象和其他类开放细节，因此这就牵扯到了一个宽接口和窄接口的问题，此问题可以由黑箱备忘录模式来解决。

22.2.2 备忘录模式黑箱实现

黑箱模式可以很好地解决备忘录类要对原生器类开放宽接口而对负责人类提供窄接口的问题，在看最终实现方法之前，先来看一下双接口的问题。

在Java中支持一个类实现多个接口，类图如图22-5所示。

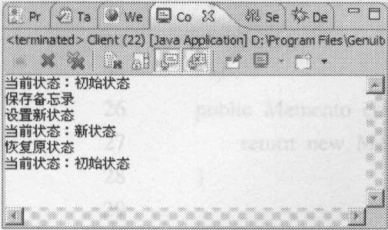


图22-4 运行结果

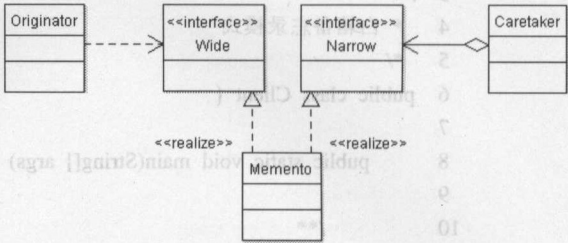


图22-5 双接口类图

通常情况下，Wide接口不会定义，而是使用Memento类本身，这种结构的问题在于：如何让Caretaker使用Narrow接口访问。
如下代码展示了内部类的作用。

代码片段7 InnerClassDemo.java

```
1 package cn.steven.pattern.demo.memento;
2
3 /**
4  * 内部类演示
5  */
6 public class InnerClassDemo {
7
8     public static void main(String[] args) {
9
10         Outer outer = new Outer();
11
12         /**
13          * 无法创建和访问内部类
14          */
15         //Outer.Inner inner;
16
17         /**
18          * 以下对象没有Inner类的任何方法，
19          * 也无法转换为Outer.Inner
20          */
21         INarrow inner = outer.getInner();
22
23     }
24
25 }
26
27 /**
28  * 声明式接口
29  */
30 interface INarrow {
31
32 }
```

```
33
34 /**
35  * 含有内部类的类
36  */
37 class Outer {
38
39     public INarrow getInner(){
40         /**
41          * 创建一个内部类对象
42          */
43         Inner inner = this.new Inner();
44         /**
45          * 内部类的私有成员可以使用
46          */
47         inner.privateMethod();
48         inner.publicMethod();
49         return inner;
50     }
51
52     private void m(){
53         System.out.println("Outer.p()");
54     }
55
56     /**
57      * 私有内部类，外部无法访问
58      */
59     private class Inner implements INarrow {
60         private void privateMethod() {
61             System.out.println("Outer.Inner.privateMethod()");
62             m();//Outer的私有成员可以使用
63         }
64
65         public void publicMethod() {
66             System.out.println("Outer.Inner.publicMethod()");
67         }
68     }
69 }
70
71 }
```

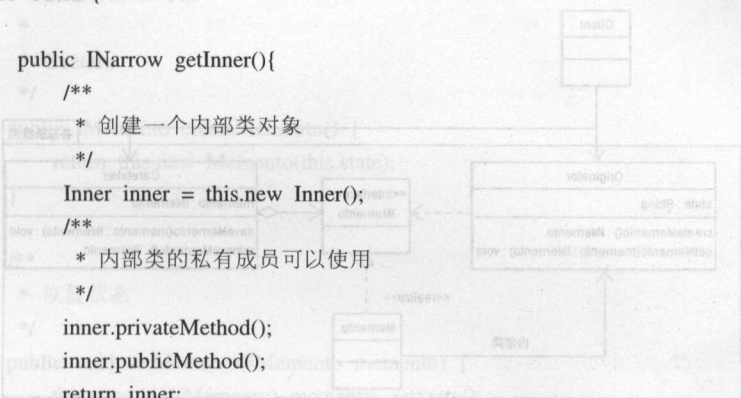


图22-7 备忘录模式类图

本段代码执行结果如图22-6所示。

在查看上述代码段时，如果没有对内部类的了解，是不容易得出代码运行的结果的。在对这段代码有疑问时可查看关于Java Inner Class和面向对象的相关知识。

这段代码给了我们一个很好的思路，如果将备忘录类作为原发器类的私有内部类并且使用声明式接口，那么外界类是不可能了解备忘录类的细节的，这样，对外就很好地隐藏了备忘录的细节，也就是对外实现了窄接口，且原

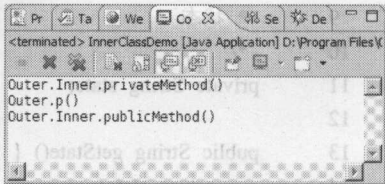


图22-6 内部类演示结果

发器类的细节也可以设置为私有类来进行隐藏。由于备忘录类和原发器类具有的这种亲密关系，它们可以相互访问各自的私有成员，这样就实现了宽接口。

类图如图22-7所示。

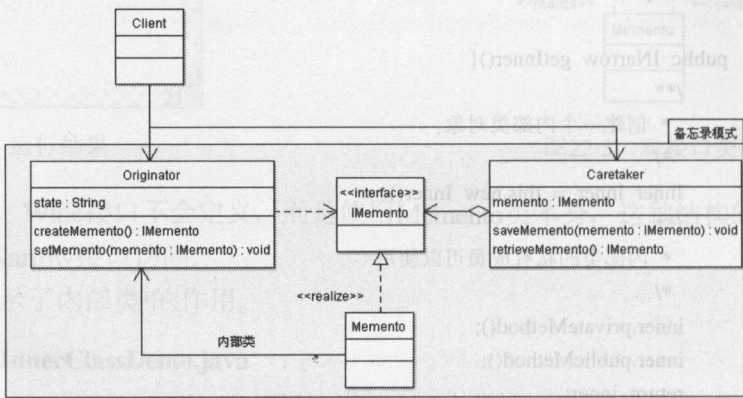


图22-7 备忘录模式类图（黑箱）

实现代码如下，声明式接口：

代码片段8 IMemento.java

```
1 package cn.steven.pattern.demo.memento.patternB;
2
3 /**
4  * 声明式接口，无需定义任何方法
5  */
6 public interface IMemento {
7
8 }
```

原生器类和内部类：

代码片段9 Originator.java

```
1 package cn.steven.pattern.demo.memento.patternB;
2
3 /**
4  * 原发器
5  */
6 public class Originator {
7
8     /**
9      * 状态
10     */
11     private String state;
12
13     public String getState() {
14         return state;
15     }
16
17     public void setState(String state) {
```

```
18     this.state = state;
19 }
20
21 /**
22  * 创建备忘录对象
23  *
24  * @return
25  */
26 public IMemento createMemento() {
27     return this.new Memento(this.state);
28 }
29
30 /**
31  * 恢复状态
32  */
33 public void setMemento(IMemento memento) {
34     this.state = ((Memento) memento).getState();
35 }
36
37 /**
38  * 备忘录，内部类
39  */
40 private class Memento implements IMemento {
41
42     /**
43      * 保存的状态
44      */
45     private String state;
46
47     public String getState() {
48         return state;
49     }
50
51     public void setState(String state) {
52         this.state = state;
53     }
54
55     /**
56      * 构造方法
57      *
58      * @param state
59      */
60     public Memento(String state) {
61         super();
62         this.state = state;
63     }
64
65 }
66
67 }
```

负责人类:

代码片段10 Caretaker.java

```
1 package cn.steven.pattern.demo.memento.patternB;
2
3 /**
4  * 负责人
5  */
6 public class Caretaker {
7
8     /**
9      * 备忘录属性
10     */
11     private IMemento memento;
12
13     /**
14      * 保存备忘录对象
15     */
16     public void saveMemento(IMemento memento) {
17         this.memento = memento;
18     }
19
20     /**
21      * 取回备忘录对象
22     */
23     public IMemento retrieveMemento() {
24         return memento;
25     }
26 }
```

客户端代码:

代码片段11 Client.java

```
1 package cn.steven.pattern.demo.memento.patternB;
2
3 /**
4  * 黑箱备忘录模式
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 创建源发起器
12          */
13         Originator originator = new Originator();
14
15         /**
16          * 创建负责人
17          */
18         Caretaker caretaker = new Caretaker();
19
20         /**
```



```

21  * 设置状态
22  */
23  originator.setState("初始状态");
24
25  /**
26  * 测试备忘录操作
27  */
28  System.out.println("当前状态: " + originator.getState());
29  System.out.println("保存备忘录");
30  caretaker.saveMemento(originator.createMemento());
31  System.out.println("设置新状态");
32  originator.setState("新状态");
33  System.out.println("当前状态: " + originator.getState());
34  System.out.println("恢复原状态");
35  originator.setMemento(caretaker.retrieveMemento());
36  System.out.println("当前状态: " + originator.getState());
37  }
38
39  }

```

客户端运行结果如图22-8所示。

黑箱模式解决了白箱模式的封装不良的问题，它在实际编程中会经常用到。

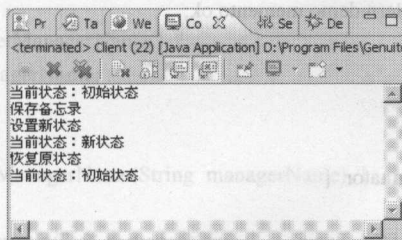


图22-8 黑箱模式运行结果

22.2.3 用备忘录模式设计可靠的数据保存系统

通过上面两节的学习，相信读者已经可以掌握简单的备忘录模式的设计方法了。通常在实际的系统应用中还会有一些变化，下面是需要思考的几个问题：

- 客户端需要同时操作原发器对象和负责人对象时，是否可以操作负责人对象？
- 是否可以支持多种原发器和备忘录？
- 是否可以保存多种属性至备忘录？
- 是否可以只保存变化的数据到备忘录，提高存储的效率？
- 是否可以用于命令模式中？

下面演示的是命令模式和备忘录模式结合使用的例子，其结构比较复杂，请读者仔细分辨架构中两种模式的结合方法，类图如图22-9所示。

注意图22-9中浅色的类为备忘录模式，深色的类为命令模式。由于结合使用了两种模式，所以备忘录和命令的细节都被屏蔽了，从而对外提供了精简的接口。

原发器接口：

代码片段10- Carotaker.java

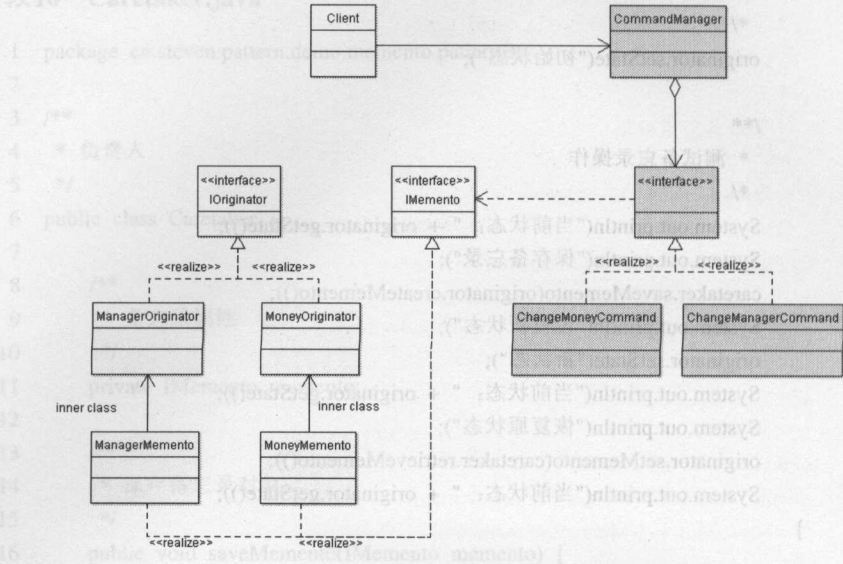


图22-9 命令模式和备忘录模式结合使用的示例类图

代码片段12 IOriginator.java

```
1 package cn.steven.pattern.demo.memento.ok;
2
3 /**
4  * 原发器接口
5  */
6 public interface IOriginator {
7     /**
8      * 创建备忘录对象
9      *
10     * @return
11     */
12     public IMemento createMemento();
13
14     /**
15      * 恢复状态
16      */
17     public void setMemento(IMemento memento);
18 }
```

经理人原发器:

代码片段13 ManagerOriginator.java

```
1 package cn.steven.pattern.demo.memento.ok;
2
3 /**
4  * 经理人信息原发器类，单例模式
5  */
6 public class ManagerOriginator implements IOriginator {
7
8     // 单例模式
9     private static ManagerOriginator instance = new ManagerOriginator();
10
11     // 构造方法
12     private ManagerOriginator() {}
13
14     // 创建备忘录对象
15     public IMemento createMemento() {
16         // ...
17     }
18
19     // 恢复状态
20     public void setMemento(IMemento memento) {
21         // ...
22     }
23 }
```

```

8  /**
9   * 采用 Initialization on Demand Holder idiom 构造单例类
10  *
11  * 内部静态类，单例持有者
12  */
13  static class SingletonHolder {
14      static ManagerOriginator instance
15          = new ManagerOriginator();
16  }
17
18  public static ManagerOriginator getInstance() {
19      return SingletonHolder.instance;
20  }
21
22  private ManagerOriginator() {
23  }
24
25
26  /**
27   * 内部状态
28   */
29  private String managerName = "目前未设置";
30
31  public String getManagerName() {
32      return managerName;
33  }
34
35  public void setManagerName(String managerName) {
36      System.out
37          .println("set managerName " + managerName);
38      this.managerName = managerName;
39  }
40
41  /**
42   * 创建备忘录
43   */
44  @Override
45  public IMemento createMemento() {
46      return this.new ManagerMemento(this);
47  }
48
49  /**
50   * 恢复备忘录
51   */
52  @Override
53  public void setMemento(IMemento memento) {
54      this.setManagerName(((ManagerMemento) memento)
55          .getState());
56  }
57
58  /**

```



```
59      * 备忘录，内部类
60      */
61      private class ManagerMemento implements IMemento {
62
63          /**
64           * 保存的状态
65           */
66          private String state;
67
68          public String getState() {
69              return state;
70          }
71
72          public void setState(String state) {
73              this.state = state;
74          }
75
76          /**
77           * 构造方法
78           *
79           * @param state
80           */
81          public ManagerMemento(ManagerOriginator originator){
82              this.state = originator.getManagerName();
83          }
84      }
85  }
86 }
```

金额原发器:

代码片段14 MoneyOriginator.java

```
1 package cn.steven.pattern.demo.memento.ok;
2
3 /**
4  * 财务信息原发器类，单例模式
5  */
6 public class MoneyOriginator implements IOiginator {
7
8     /**
9      * 采用 Initialization on Demand Holder idiom 构造单例类
10     *
11     * 内部静态类，单例持有者
12     */
13     static class SingletonHolder {
14         static MoneyOriginator instance = new MoneyOriginator();
15     }
16
17     public static MoneyOriginator getInstance() {
18         return SingletonHolder.instance;
19     }
```

```

20  * 执行命令
21  private MoneyOriginator() {
22  @Override
23  } public void execute(Object... objects) {
24  /**
25  /** * 创建备忘录
26  * 内部状态
27  */ memento = MoneyMemento(this);
28  private float money = 0;
29
30  public float getMoney() {
31      return money;
32  }
33
34  public void setMoney(float money) {
35      System.out.println("set money " + money);
36      this.money = money;
37  }
38
39  /**
40  * 创建备忘录
41  */
42  @Override
43  public IMemento createMemento() {
44      return this.new MoneyMemento(this);
45  }
46  public void undo() {
47  /** ManagerOriginator.getInstance().setMoney(memento.getState());
48  * 恢复备忘录
49  */
50  @Override
51  public void setMemento(IMemento memento) {
52      this.setMoney(((MoneyMemento) memento).getState());
53  }
54
55  /**
56  * 备忘录, 内部类
57  */
58  private class MoneyMemento implements IMemento {
59
60  /** ChangeMoneyCommand implements ICommand {
61  * 保存的状态
62  */
63  private float state;
64
65  public float getState() {
66      return state;
67  }
68
69  public void setState(float state) {
70  @Override this.state = state;

```

```

71     }
72
73     /**
74      * 构造方法
75      *
76      * @param state
77      */
78     public MoneyMemento(MoneyOriginator originator) {
79         this.state = originator.getMoney();
80     }
81
82     }
83 }

```

命令接口:

代码片段15 ICommand.java

```

1  package cn.steven.pattern.demo.memento.ok;
2
3  /**
4   * 命令接口 @param state
5   */
6  public interface ICommand {
7
8      /**
9       * 执行方法, 采用动态参数
10      */
11      void execute(Object... objects);
12
13      /**
14       * 撤销方法
15       */
16      void undo();
17 }

```

更改经理人命令:

代码片段16 ChangeManagerCommand.java

```

1  package cn.steven.pattern.demo.memento.ok;
2
3  /**
4   * 修改经理人命令
5   */
6  public class ChangeManagerCommand implements ICommand {
7
8      /**
9       * 保存备忘录
10     */
11     private IMemento memento;
12
13     /**

```



```

14     * 执行命令
15     */
16     @Override
17     public void execute(Object... objects) {
18         /**
19          * 创建备忘录
20          */
21         memento = ManagerOriginator.getInstance()
22             .createMemento();
23         /**
24          * 执行改变，这里需要传递一个String类型的参数
25          */
26         if (objects.length == 1) {
27             ManagerOriginator.getInstance().setManagerName(
28                 (String) objects[0]);
29         } else {
30             throw new RuntimeException("参数不合法!");
31         }
32     }
33
34 }
35
36 /**
37  * 撤销命令
38  */
39     @Override
40     public void undo() {
41         ManagerOriginator.getInstance().setMemento(memento);
42     }
43
44 }

```

修改金额命令:

代码片段17 ChangeMoneyCommand.java

```

1 package cn.steven.pattern.demo.memento.ok;
2
3 /**
4  * 修改金额命令
5  */
6 public class ChangeMoneyCommand implements ICommand {
7
8     /**
9      * 保存备忘录
10     */
11     private IMemento memento;
12
13     /**
14      * 执行命令
15     */
16     @Override

```

客户输入代码:

```
17 public void execute(Object... objects) {
18
19     /**
20      * 创建备忘录
21      */
22     memento = MoneyOriginator.getInstance()
23         .createMemento();
24
25     /**
26      * 执行改变，这里需要传递一个Float类型的参数
27      */
28     if (objects.length == 1) {
29         MoneyOriginator.getInstance().setMoney(
30             (Float) objects[0]);
31     } else {
32         throw new RuntimeException("参数不合法！");
33     }
34 }
35
36
37 /**
38  * 撤销命令
39  */
40 @Override
41 public void undo() {
42     MoneyOriginator.getInstance().setMemento(memento);
43 }
44
45 }
```

命令管理器：

代码片段18 CommandManager.java

```
1 package cn.steven.pattern.demo.memento.ok;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 /**
7  * 命令管理类，等同于负责人，单例模式
8  */
9 public class CommandManager {
10
11     /**
12      * 采用 Initialization on Demand Holder idiom 构造单例类
13      *
14      * 内部静态类，单例持有者
15      */
16     static class SingletonHolder {
17         static CommandManager instance = new CommandManager();
18     }
```

```

19
20 public static CommandManager getInstance() {
21     return SingletonHolder.instance;
22 }
23
24 private CommandManager() {
25
26 }
27
28 /**
29  * 保存命令序列
30  */
31 private List<ICommand> commandList = new LinkedList<ICommand>();
32
33 /**
34  * 执行命令
35  */
36 public void execute(ICommand command, Object... objects) {
37     System.out.println("执行命令" + command);
38     /**
39      * 执行
40      */
41     command.execute(objects);
42     /**
43      * 保存
44      */
45     commandList.add(command);
46 }
47
48 /**
49  * 取消命令
50  */
51 public void undo() {
52     if (commandList.size() > 0) {
53         /**
54          * 移除最后一条命令
55          */
56         ICommand command = commandList
57             .remove(commandList.size() - 1);
58         /**
59          * 取消命令
60          */
61         System.out.println("取消命令" + command);
62         command.undo();
63     } else {
64         System.out.println("命令队列中已空");
65     }
66 }
67 }
68 }

```

客户端代码:

代码片段19 Client.java

```
1 package cn.steven.pattern.demo.memento.ok;
2
3 /**
4  * 备忘录模式+命令模式
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 创建负责人（命令管理器）
12          */
13         CommandManager commandManager = CommandManager
14             .getInstance();
15
16         /**
17          * 执行命令
18          */
19         commandManager.execute(new ChangeManagerCommand(),
20             "包公");
21         commandManager.execute(new ChangeMoneyCommand(),
22             555f);
23
24         /**
25          * 回退命令
26          */
27         commandManager.undo();
28         commandManager.undo();
29         commandManager.undo();
30
31     }
32 }
```

代码运行结果如图22-10所示。

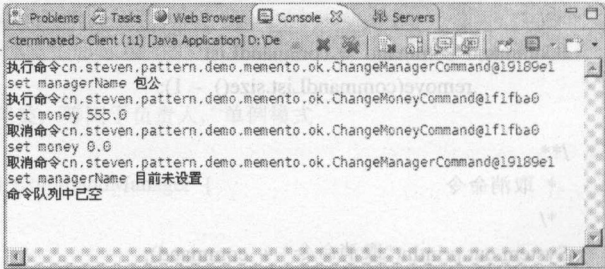


图22-10 运行结果

由结果可见，两种模式已经成功地组合在一起了。在实际问题的解决过程中，经常会见到多种模式组合使用的情况，请读者仔细斟酌使用。

22.2.4 备忘录模式在JDK中的实例

Java中备忘录模式的实现需要使用到内部类的概念，而C++中的实现可以使用友元类的概念，读者需要了解Java中与OO相关的知识才能明白其实现原理。

在软件编程中有很多方面可以用到备忘录模式，比如运行一个软件时进行了各种参数的设置，当下一次再运行此程序时需要保留上一次的设置情况，在实现这种功能时通常会使用XML¹（可扩展置标语言（eXtensible Markup Language，简称XML），是一种置标语言。）作为外部的一个状态存储配置文件。

在Web应用中，通常会使用HTTP²（HyperText Transfer Protocol，超文本传输协议）作为传输协议，但是这种协议有一种特性，就是不会存储客户端的状态，这样带来的问题就是如果一个用户在一个网站上进行了多种操作，最后网站只会知道客户的最后一种操作，如此一来，购物车的概念就没法实现了。在JavaEE³中，提供了几种方案以解决HTTP会话的问题，一种典型的操作方法就是Session。

```
public interface javax.servlet.http
```

```
HttpSession
```

HttpSession提供一种方式，可以跨多个页面请求或对Web站点的多次访问来标识用户并存储有关该用户的信息。

servlet容器使用HttpSession创建HTTP客户端和HTTP服务器之间的会话。会话将保留指定的时间段，此时间段的长度可以通过servlet容器制定，它可以为会话用户保存一定量的数据，这些数据可以在用户访问同一容器的多个资源时使用。一个会话通常对应于一个用户，该用户可以多次访问一个站点。服务器能够以多种方式维护会话，比如使用cookie或重写URL。

HttpSession允许servlet容器执行以下操作：

- 查看和操作有关某个会话的信息，比如会话标识符、创建时间和最后一次访问时间；
- 将对象绑定到会话，允许跨多个用户连接来保留用户信息。

当应用程序将对象存储到会话中或从会话中移除对象时，该会话将检查对象是否实现了HttpSessionBindingListener。如果实现了，则servlet将通知该对象它已经被绑定到会话，或者已从会话中取消对它的绑定。通知是在绑定方法完成后发送的。对于无效或过期的会话，通知是在会话已经无效或过期之后发送的。

当servlet容器使用分布式容器的设置在VM之间迁移会话时，所有实现HttpSessionActivationListener接口的会话属性都会得到通知。

servlet应该能够处理客户端选择不加入会话的情况，比如故意关闭Cookie时。在客户端加入会话前，isNew一直返回true值。如果客户端选择不加入会话，则getSession将对每个请求返回一个不同的会话，并且isNew将总是返回true值。

会话信息的范围仅限于当前Web应用程序（ServletContext），因此存储在一个上下文

¹<http://zh.wikipedia.org/wiki/XML>。

²<http://zh.wikipedia.org/zh-cn/Http>。

³http://zh.wikipedia.org/wiki/Java_EE。

中的信息在另一个上下文中不是直接可见的。

另一种典型的保存会话状态的组件是Cookie。

```
public class javax.servlet.http
```

```
Cookie
```

```
implements: Cloneable
```

Cookie是servlet发送到Web浏览器的少量信息，这些信息由浏览器保存，然后发送回服务器。Cookie的值可以唯一地标识客户端，因此Cookie常用于会话管理。

一个Cookie拥有一个名称、一个值和一些可选属性，比如注释、路径和域限定符、最大生存时间和版本号。一些Web浏览器在处理可选属性方面存在bug，因此有节制地使用这些属性可提高servlet的互操作性。

servlet通过使用`HttpServletResponse#addCookie`方法将Cookie发送到浏览器，该方法将字段添加到HTTP响应头，以便一次一个地将Cookie发送到浏览器。浏览器应该支持每台Web服务器可以有20个Cookie，总共可以有300个Cookie，并且可将每个Cookie的大小限定为4KB。

浏览器通过向HTTP请求头添加字段将Cookie返回给servlet。可使用`HttpServletRequest#getCookies`方法从请求中获取Cookie。一些Cookie可能有相同的名称，但却有不同的路径属性。

Cookie影响使用它们的Web页面的缓存。HTTP 1.0不会缓存那些使用Cookie类创建的Cookie的页面。Cookie类不支持HTTP 1.1中定义的缓存控件。

Cookie类支持版本0（遵守Netscape协议）和版本1（遵守RFC 2109协议）Cookie规范。默认情况下，Cookie是使用版本0创建的，这样可以确保具有最佳互操作性。

通过上面的说明可见，保存状态在Web应用程序中更为重要，选择合适的状态管理方法在B/S¹架构的应用程序开发中是一个重要的环节。

22.2.5 备忘录模式的特点

备忘录模式有以下特点：

- 保持封装的边界。使用备忘录可以避免暴露一些只应由原发器管理却又必须存储在原发器之外的信息。该模式把可能很复杂的Originator内部信息对其他对象屏蔽起来，从而保持了封装的边界。

- 它简化了原发器。在其他的保持封装性的设计中，Originator负责保持客户请求过的内部状态版本。这就把所有存储管理的重任交给了Originator。让客户管理他们请求的状态将会简化Originator的工作，并且使得客户工作结束时无需通知原发器。

- 使用备忘录可能代价很高。如果原发器在生成备忘录时必须拷贝并存储大量的信息，或者客户非常频繁地创建备忘录和恢复原发器状态，可能会导致非常大的开销。除非封装和恢复Originator状态的开销不大，否则该模式可能并不合适。

- 在一些语言中定义窄接口和宽接口可能难以保证只有原发器可访问备忘录的状态。

¹<http://zh.wikipedia.org/zh-cn/%E6%B5%8F%E8%A7%88%E5%99%A8-%E6%9C%8D%E5%8A%A1%E5%99%A8>.

• 维护备忘录的潜在代价是由管理器负责删除它所维护的备忘录，然而管理器不知道备忘录中有多少个状态，因此当存储备忘录时，一个本来很小的管理器，可能会产生大量的存储开销。

备忘录模式的优、缺点：

使用备忘录模式来保存对象的历史状态可以有效地保持封装边界。使用备忘录可以避免暴露一些只应由“备忘发起角色”管理却又必须存储在“备忘发起角色”之外的信息。同时可以把“备忘发起角色”的内部信息对其他对象屏蔽起来，从而保持了封装边界。但是如果备份的“备忘发起角色”存在大量的信息或者创建、恢复操作非常频繁，则可能造成很大的开销。

22.2.6 与其他模式的关系

备忘录模式通常可以和命令模式结合使用，以便为后者提供保存状态的功能。当备忘录模式中需要设置多个检查点时（多次备份），通常需要使用迭代器模式来遍历多个检查点。

22.3 备忘录模式总结

在讲命令模式的时候，我们曾经提到利用中间的命令角色可以实现undo、redo的功能。从备忘录模式的定义可以看出备忘录模式是专门用来存放对象历史状态的，这对于很好地实现undo、redo功能有很大的帮助。所以在命令模式中undo、redo功能可以配合备忘录模式来实现。

在实际编程中，如果遇到要恢复状态的问题，就可以考虑使用备忘录模式。

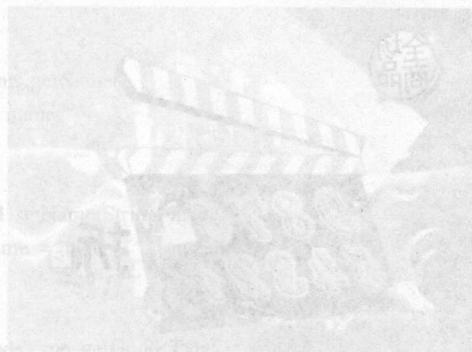


图23-1 备忘录模式示意图

```
29 this.price = price;
30 this.name = name;
31 }
32
33 public String getName() {
34     return this.name;
35 }
36
37 public void setName(String name) {
38     this.name = name;
39 }
40
41 public GoodType getGoodType() {
42     return goodType;
43 }
44
45 public void setGoodType(GoodType goodType) {
46     this.goodType = goodType;
47 }
48
49 package cn.steven.pattern.demo.strategy;
50 import java.util.*;
51
52 public class Strategy {
53     private List<Strategy> strategies;
```

第23章 策略模式 (Strategy)

策略模式属于行为模式，其目的是定义一系列算法，把它们一个个封装起来，并且使它们可以互相替换，从而使得算法可以独立于使用它的客户而发生变化¹。

比如在制做文字输入软件的时候，需要对已经输入的文本进行换行，有很多的算法可以完成这种功能，比如每隔100个字符进行换行，使用\n风格换行或者使用\r\n换行等，将这些算法代码硬编码进使用它们的类中是不可取的，其原因如下：

- 需要使用换行功能的客户程序如果直接包含换行算法代码的话将会变得很复杂，这使得客户程序大并且难以维护，尤其是当它需要支持多种换行算法时会更为严重（如23.1实例所示）。
- 如果将换行功能代码存储于具体需要使用该功能的类中，将会导致类的换行功能无法修改。
- 当换行功能是客户程序的一个难以分割的成分时，增加新的换行算法或改变现有算法将十分困难。

策略模式的主要思想是定义一些包含算法的类来封装不同的换行算法，从而解决修改和扩展的问题。

23.1 最常用的促销手段——打折

商业销售中常见的促销手段就是打折，在不同的时期对于不同的商品，销售方通常会采取不同的折扣方式，比如我们设计的超市现在就需要开展促销活动，如图23-1所示。

其中服装打8折，食品打9折，家电不打折。



图23-1 促销活动示意图

下面来看一下具体的实现代码。
首先看一下商品类，此类具有“商品类型”、“商品名称”和“商品价格”属性：

```
代码片段1 Goods.java
1 package cn.steven.pattern.demo.strategy.quest;
```

¹Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.GOF[95]

```

2
3 /**
4  * 商品类
5  */
6 public class Goods {
7     /**
8      * 商品的类型
9      */
10    private GoodsType goodsType;
11
12    /**
13     * 商品价格
14     */
15    private float price;
16
17    /**
18     * 商品名称
19     */
20    private String name;
21
22    /**
23     * 构造方法
24     */
25    public Goods(GoodsType goodsType, String name,
26                float price) {
27        super();
28        this.goodsType = goodsType;
29        this.price = price;
30        this.name = name;
31    }
32
33    public String getName() {
34        return name;
35    }
36
37    public void setName(String name) {
38        this.name = name;
39    }
40
41    public GoodsType getGoodsType() {
42        return goodsType;
43    }
44
45    public void setGoodsType(GoodsType goodsType) {
46        this.goodsType = goodsType;
47    }
48
49    public float getPrice() {
50        return price;
51    }
52
53    public void setPrice(float price) {

```



```
54         this.price = price;
55     }
56
57 }
```

注意在代码片段1第10行出现了一个枚举¹（ENUM）类型，这种类型是在JDK 1.5版本之后出现的Java新特性。枚举类型的出现使程序员无需再定义常量的一些属性。

如下是枚举类：

代码片段2 GoodsType.java

```
1 package cn.steven.pattern.demo.strategy.quest;
2
3 /**
4  * 商品类型枚举
5  */
6 public enum GoodsType {
7     服装,
8     食品,
9     家电
10 }
```

客户端代码：

代码片段3 Client.java

```
1 package cn.steven.pattern.demo.strategy.quest;
2
3 /**
4  * 客户端代码
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 下面定义的打折策略是：
12          * 服装：8折
13          * 食品：9折
14          * 家电：不打折
15          */
16
17         Goods g1 = new Goods(GoodsType.家电, "冰箱", 998f);
18         Goods g2 = new Goods(GoodsType.服装, "西装", 552f);
19         Goods g3 = new Goods(GoodsType.食品, "面包", 4.5f);
20
21         Client.showPrice(g1);
22         Client.showPrice(g2);
23         Client.showPrice(g3);
24     }
```

¹<http://java.sun.com/docs/books/tutorial/java/javaOO/enum.html>.

```

25
26 public static void showPrice(Goods goods) {
27     double price = 0;
28
29     switch (goods.getGoodsType()) {
30     case 家电:
31         price = goods.getPrice();
32         break;
33     case 服装:
34         price = goods.getPrice() * .8;
35         break;
36     case 食品:
37         price = goods.getPrice() * .9;
38         break;
39     default:
40         price = goods.getPrice();
41         break;
42     }
43
44     System.out.println(goods.getName() + "["
45         + goods.getGoodsType() + "]" + "\t原价: "
46         + goods.getPrice() + "\t现价: " + price);
47 }
48
49 }

```

注意代码片段3中第29~42行使用了switch结构来判断打折的策略。

代码运行结果如图23-2所示。

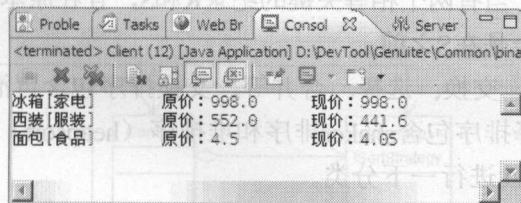


图23-2 Client.java运行结果

由结果可见，通过这种编程方式的确可以完成打折的功能，但是如果打折的规则非常多，则此种编程方式会产生大量的耦合度极高的代码，而且很明显，这类代码还会经常地被修改。

本章介绍的策略模式将会很好地解决此类多算法的问题。

23.2 策略模式的结构

策略模式将会把算法独立成一个类层次，读者在分析策略模式时要注意这一点。

23.2.1 策略模式

下面演示策略模式的实现方法，示例为对数组进行排序的算法。对数据排序¹是一个经典

¹<http://zh.wikipedia.org/wiki/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95>。

的算法问题，因为有多种算法。

在下述内容中出现了一个算法复杂度的概念，通常记为 $O(x)$ ，如果需要验证一个计算的算法是否有效，时间和空间是最重要的两项资源，时间就是计算机运行的复杂度，空间就是计算机计算时所需要的额外存储空间的复杂度。

用时间作为例子来讨论算法分析的过程如下：如果将输入的长度（设为 n ）作为变量，而关注的是算法运行时间关于 n 的函数关系 $T(n)$ 。因为一个算法在不同的计算模型上实现时， $T(n)$ 可能会有常数因子的差别，使用 O 表达式来表示 $T(n)$ ，就可以忽略在不同计算模型上实现的常数因子。

以搜索这个计算任务为例。在搜索问题中，给定了一个具体的数 s ，和长度为 n 的数组 A （数组中数的位置用1到 n 做标记），任务是当 s 在 A 中时，找到 s 的位置，而 s 不在 A 中时，需要报告“未找到”。这时输入的长度即为 $n+1$ 。下面的过程即是一个最简单的算法：依次扫描 A 中的每个数，并与 s 进行比较，如果相等即返回当前的位置，如果扫描所有的数而算法仍未停止，则返回“未找到”。如果假设 s 在 A 中每个位置都是等可能的，那么算法在找到 s 的条件下需要 $1/n (1+2+\dots+n)=n(n+1)/2n=(n+1)/2$ 的时间。如果 s 不在 A 中，那么需要 $(n+1)$ 的时间。由 O 表达式的知识可以知道这个算法所需的时间即为 $O(n)$ 。

在计算机科学所使用的排序算法通常被分类为：

- 计算的复杂度（最差、平均和最好表现），依据串行（list）的大小（ n ）。一般而言，好的表现是 $O(n \log n)$ ，且坏的行为是 $O(n^2)$ 。对于一个排序理想的表现是 $O(n)$ 。仅使用一个抽象关键比较运算的排序算法总平均上总是至少需要 $O(n \log n)$ 。

- 内存使用量（以及其他电脑资源的使用）。

- 稳定度：稳定排序算法会依照相等的关键（换言之就是值）维持纪录的相对次序。也就是一个排序算法是稳定的，当有两个相等关键的纪录 R 和 S ，且在原本的串行中 R 出现在 S 之前，在排序过的串行中 R 也将会是在 S 之前。

- 一般的方法：插入、交换、选择、合并等。交换排序包含冒泡排序（bubble sort）和快速排序（quicksort）。选择排序包含shaker排序和堆排序（heapsort）。

下面按照算法的稳定性进行一下分类。

稳定的排序算法：

- 泡沫排序（bubble sort）——时间复杂度 $O(n^2)$ ；
- 鸡尾酒排序（cocktail sort，双向的冒泡排序）——时间复杂度 $O(n^2)$ ；
- 插入排序（insertion sort）——时间复杂度 $O(n^2)$ ；
- 桶排序（bucket sort）——时间复杂度 $O(n)$ ；需要 $O(k)$ 额外空间；
- 计数排序（counting sort）——时间复杂度 $O(n+k)$ ；需要 $O(n+k)$ 额外空间；
- 合并排序（merge sort）——时间复杂度 $O(n \log n)$ ；需要 $O(n)$ 额外空间；
- 原地合并排序——时间复杂度 $O(n^2)$ ；
- 二叉排序树排序（binary tree sort）——时间复杂度 $O(n \log n)$ 为期望性能；时间复杂度 $O(n^2)$ 为最坏性能；需要 $O(n)$ 额外空间；
- 鸽巢排序（pigeonhole sort）——时间复杂度 $O(n+k)$ ；需要 $O(k)$ 额外空间；
- 基数排序（radix sort）——时间复杂度 $O(n \cdot k)$ ；需要 $O(n)$ 额外空间。

不稳定的排序算法:

- 选择排序 (selection sort) ——时间复杂度 $O(n^2)$;
- 希尔排序 (shell sort) ——时间复杂度 $O(n \log n)$;
- 堆排序 (heapsort) ——时间复杂度 $O(n \log n)$;
- 快速排序 (quicksort) ——时间复杂度 $O(n \log n)$ 期望时间, 时间复杂度 $O(n^2)$ 最坏情况;

对于大的、乱数串行一般是最快的已知排序;

- Introsort——时间复杂度 $O(n \log n)$;
- Patience sorting——时间复杂度 $O(n \log n+k)$ 最坏情况时间, 需要额外的 $O(n+k)$ 空间, 也需要找到最长的递增子序列 (longest increasing subsequence)。

不实用的排序算法:

- Bogo排序——时间复杂度 $O(n \times n!)$ 期望时间, 无穷的最坏情况;
- Stupid sort——时间复杂度 $O(n^3)$; 递归版本需要 $O(n^2)$ 额外内存;
- Bead sort——时间复杂度 $O(n)$ 或时间复杂度 $O(\sqrt{n})$, 但需要特别的硬件;
- Pancake sorting——时间复杂度 $O(n)$, 但需要特别的硬件。

以下是对数组数据进行排序的建议:

- 数组长度不大的情况下不宜使用合并排序, 其他排序方法的效果差别不大。
- 数组长度很大的情况下希尔排序最快, 快速排序其次, 冒泡最慢。

由以上内容可见, 不同的排序方法所需要的辅助空间和时间复杂度是不同的, 通常计算时倾向于采用速度快的, 也就是时间复杂度低的, 但是在有外部限制的某些情况下, 比如在嵌入式设备中, 算法的空间复杂度也是有要求的, 这就需要程序员选用合适的算法。

下面为了演示策略模式, 对数组数据使用了冒泡排序¹、快速排序²和希尔排序³, 设计类图如图23-3所示。

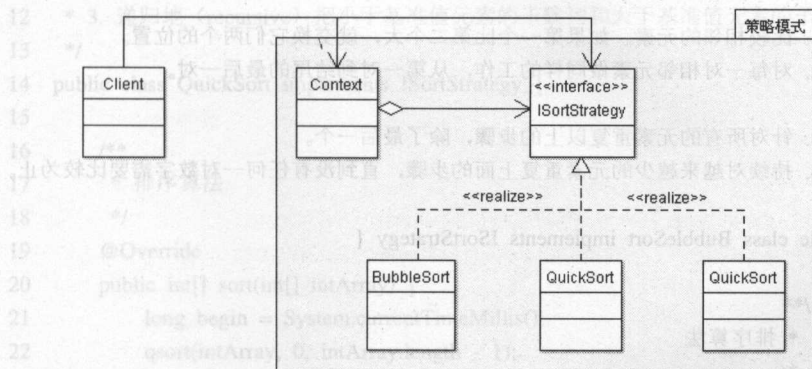


图23-3 策略模式类图

图中的参与者如下:

- 环境 (Context) 角色: 持有一个策略 (Strategy) 角色的引用。也叫上下文。

¹<http://zh.wikipedia.org/zh-cn/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F>.

²<http://zh.wikipedia.org/zh-cn/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F>.

³<http://zh.wikipedia.org/zh-cn/%E5%B8%8C%E5%B0%94%E6%8E%92%E5%BA%8F>.

- 抽象策略（Strategy）角色：这是一个抽象角色，通常由一个接口或一个抽象类来实现。
- 具体策略（xxxSort）角色：包装了相应的算法和行为。
- 客户端（Client）：使用Context和具体的策略对象进行操作。

策略接口如下：

代码片段4 ISortStrategy.java

```
1 package cn.steven.pattern.demo.strategy.pattern;  
2  
3 /**  
4  * 排序策略接口  
5  */  
6 public interface ISortStrategy {  
7     /**  
8      * 排序操作  
9      *  
10     * @param intArray  
11     * @return  
12     */  
13     int[] sort(int[] intArray);  
14 }
```

冒泡排序策略：

代码片段5 BubbleSort.java

```
1 package cn.steven.pattern.demo.strategy.pattern;  
2  
3 /**  
4  * 冒泡排序，冒泡排序算法的运行步骤如下：  
5  *  
6  * 1. 比较相邻的元素。如果第一个比第二个大，就交换它们两个的位置。  
7  * 2. 对每一对相邻元素做同样的工作，从第一对到结尾的最后一对。  
8  *  
9  * 3. 针对所有的元素重复以上的步骤，除了最后一个。  
10 * 4. 持续对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较为止。  
11 */  
12 public class BubbleSort implements ISortStrategy {  
13  
14     /**  
15      * 排序算法  
16      */  
17     @Override  
18     public int[] sort(int[] intArray) {  
19  
20         long begin = System.currentTimeMillis();  
21         bubbleSort(intArray);  
22         System.out.println("BubbleSort:"  
23             + (System.currentTimeMillis() - begin) + "ms");  
24  
25         return intArray;  
26     }
```

快速排序策略:

```

1 package cn.steven.pattern.demo.strategy.pattern;
2
3 /**
4  * 快速排序
5  * 快速排序使用分治法（divide and conquer）策略来把一个序列（list）
6  * 分为两个子序列（sub-lists）。
7  * 步骤为：
8  * 1. 从数列中挑出一个元素，称为“基准”（pivot）。
9  * 2. 重新排序数列，所有比基准值小的元素摆放在基准前面，
10  * 所有比基准值大的元素摆在基准的后面（相同的数可以放到任意一边）。这个
11  * 称为分割（partition）操作。
12  * 3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列进行排序。
13  */
14 public class QuickSort implements ISortStrategy {
15
16     /**
17      * 排序算法
18      */
19     @Override
20     public int[] sort(int[] intArray) {
21         long begin = System.currentTimeMillis();
22         qsort(intArray, 0, intArray.length - 1);
23         System.out.println("QuickSort:"
24             + (System.currentTimeMillis() - begin) + "ms");
25         return intArray;
26     }
27
28     /**
29      * 快速排序
30      * @param array
31      * @param begin
32      * @param end
33      */

```



```
34 private void qsort(int[] array, int begin, int end) {
35     if (end > begin) {
36         int index = partition(array, begin, end);
37         qsort(array, begin, index - 1);
38         qsort(array, index + 1, end);
39     }
40 }
41
42 /**
43  * 交换值
44  * @param array
45  * @param i
46  * @param j
47  */
48 private void swap(int[] array, int i, int j) {
49     int tmp = array[i];
50     array[i] = array[j];
51     array[j] = tmp;
52 }
53
54 /**
55  * 数组分割处理
56  * @param array
57  * @param begin
58  * @param end
59  * @return
60  */
61 private int partition(int[] array, int begin, int end) {
62     /**
63      * 二分法
64      */
65     int index = begin + (end - begin) / 2;
66     /**
67      * 对比值
68      */
69     int pivot = array[index];
70     swap(array, index, end);
71     for (int i = begin; i < end; ++i) {
72         if (array[i] < pivot) {
73             swap(array, index++, i);
74         }
75     }
76     swap(array, index, end);
77     return (index);
78 }
79
80 }
```

希尔排序策略:

代码片段7 ShellSort.java

```
1 package cn.steven.pattern.demo.strategy.pattern;
```

```
2
3 /**
4  * 希尔排序
5  * 对有n个元素的数组，先取一个小于n的整数d1作为第一个增量，把文件的全部
6  * 记录分成d1个组。所有距离为d1的倍数的记录放在同一个组中。先在各组内进行直接
7  * 插入排序；然后，取第二个增量d2<d1重复上述的分组和排序，直至所取的增量
8  *  dt=1(dt<dt-1<...<d2<d1)，即所有记录放在同一组中进行直接插入排序为止。
9  *
10 * 该方法实质上是一种分组插入方法
11 */
12 public class ShellSort implements ISortStrategy {
13
14     /**
15      * 排序方法
16      */
17     @Override
18     public int[] sort(int[] intArray) {
19         long begin = System.currentTimeMillis();
20         shellsort(intArray, intArray.length);
21         System.out.println("ShellSort:"
22             + (System.currentTimeMillis() - begin) + "ms");
23         return intArray;
24     }
25
26     /**
27      * 希尔排序
28      *
29      * @param a
30      * @param n
31      */
32     private void shellsort(int[] a, int n) {
33         int i, j, k, temp, gap;
34         /**
35          * 增量
36          */
37         int[] gaps = { 1, 5, 13, 43, 113, 297, 815, 1989, 4711,
38             11969, 27901, 84801, 213331, 543749, 1355339,
39             3501671, 8810089, 21521774, 58548857, 157840433,
40             410151271, 1131376761, 2147483647 };
41         for (k = 0; gaps[k] < n; k++) {
42             // 空
43         }
44         while (--k >= 0) {
45             gap = gaps[k];
46             for (i = gap; i < n; i++) {
47                 temp = a[i];
48                 j = i;
49                 while (j >= gap && a[j - gap] > temp) {
50                     a[j] = a[j - gap];
51                     j = j - gap;
52                 }
53                 a[j] = temp;
```

```
54 private void qsort(int[] array, int begin, int end) {
55     if (end > begin) {
56         int index = partition(array, begin, end);
57         qsort(array, begin, index);
58         qsort(array, index + 1, end);
59     }
60 }
```

上下文类:

代码片段8 Context.java

```
1 package cn.steven.pattern.demo.strategy.pattern;
2
3 /**
4  * 策略模式的上下文类
5  */
6 public class Context {
7     // 聚合排序对象
8     private ISortStrategy sortStrategy;
9
10    public ISortStrategy getSortStrategy() {
11        return sortStrategy;
12    }
13
14    public void setSortStrategy(ISortStrategy sortStrategy) {
15        this.sortStrategy = sortStrategy;
16    }
17
18    /**
19     * 构造方法，传入一个排序策略
20     * @param sortStrategy
21     */
22    public Context() {
23        // 初始化排序策略
24        this.setSortStrategy(new ShellSort());
25    }
26
27    /**
28     * 执行排序
29     */
30    public int[] sort(int[] intArray) {
31        this.intArray = this.getSortStrategy().sort(intArray);
32        return this.intArray;
33    }
34
35    /**
36     * 排序数组
37     */
38    private int[] intArray;
39
40    public int[] getIntArray() {
41        return intArray;
42    }
43 }
```



```

44     }
45
46     public void setIntArray(int[] intArray) {
47         this.intArray = intArray;
48     }
49
50 }

```

客户类:

代码片段9 Client.java

```

1  package cn.steven.pattern.demo.strategy.pattern;
2
3  import java.util.Random;
4
5  /**
6   * 排序策略模式客户端代码
7   */
8  public class Client {
9
10     public static void main(String[] args) {
11         /**
12          * 创建上下文
13          */
14         Context context = new Context();
15
16         /**
17          * 测试各种排序
18          */
19         testBubbleSort(context);
20         testQuickSort(context);
21         testShellSort(context);
22     }
23
24     public static Random rand = new Random();
25
26     /**
27      * 随机数组大小
28      */
29
30     public static int arraySize = 15;
31
32     /**
33      * 随机数字大小
34      */
35
36     public static int maxNum = 100;
37
38     /**
39      * 生成测试数据
40      */
41     public static int[] makeRandomArray() {
42         int[] intArray = new int[arraySize];

```

```

42         for (int i = 0; i < arraySize; i++) {
43             intArray[i] = rand.nextInt(maxNum);
44         }
45         return intArray;
46     }
47
48     public static void testBubbleSort(Context context) {
49         int[] intArray = makeRandomArray();
50         context.setSortStrategy(new BubbleSort());
51         intArray = context.sort(intArray);
52         System.out.print("冒泡排序: ");
53         print(intArray);
54     }
55
56     public static void testQuickSort(Context context) {
57         int[] intArray = makeRandomArray();
58         context.setSortStrategy(new QuickSort());
59         intArray = context.sort(intArray);
60         System.out.print("快速排序: ");
61         print(intArray);
62     }
63
64     public static void testShellSort(Context context) {
65         int[] intArray = makeRandomArray();
66         context.setSortStrategy(new ShellSort());
67         intArray = context.sort(intArray);
68         System.out.print("希尔排序: ");
69         print(intArray);
70     }
71
72     /**
73      * 打印数组
74      *
75      * @param intArray
76      */
77     public static void print(int[] intArray) {
78         for (int i = 0; i < intArray.length; i++) {
79             System.out.print(intArray[i] + " ");
80         }
81         // System.out.print("结果略");
82         System.out.println();
83     }
84
85 }

```

运行结果如图23-4所示。

由图23-4可见，使用任何一种排序都可以得到正确的结果。在上面的代码中还实现了一个统计运行时间的功能，在运行结果中没有显示出来，因为数据量太小，所以各种排序的效率没有太大差异，如图23-5是采用50000条记录排序的结果。

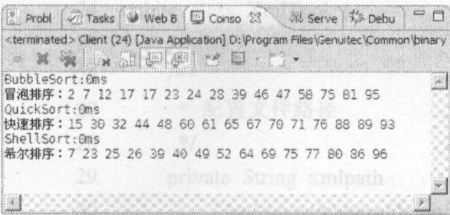


图23-4 排序运行结果

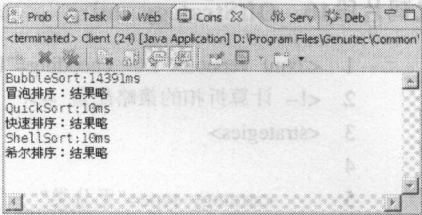


图23-5 排序效率对比

由图23-5可见，在数据量较大的情况下，快速排序和希尔排序显然要比冒泡排序的效率高，所以在实际编程中，程序员一定要根据具体需求选择合适的算法，算法选择不当将会造成巨大的性能问题。

23.2.2 用策略模式构建灵活的打折方式

由前面讲述的内容可以发现，策略模式和状态模式的类图几乎一样，在具体应用中这两种模式也经常会令程序员感到不解，为了说明其不同之处，下面的具体案例采取了状态模式和策略模式同时使用的方式来进行讲解。

在超市打折活动中通常会有两种打折方法，一种是同一折扣，比如某天8:00~9:00全部商品9折；再一种就是分类打折，比如服装类8折，玩具类9折这种。具有灵活的打折方式就要把这两方种方法全都实现出来，但是很明显方法1和方法2的功能性不同。经过分析，我们决定将方法1用状态模式实现，状态指的就是商店当前的状态，状态是经常改变的。方法2使用策略模式来实现，策略通常有针对性，易替换，通常选定后不变。

设计的类图如图23-6所示。

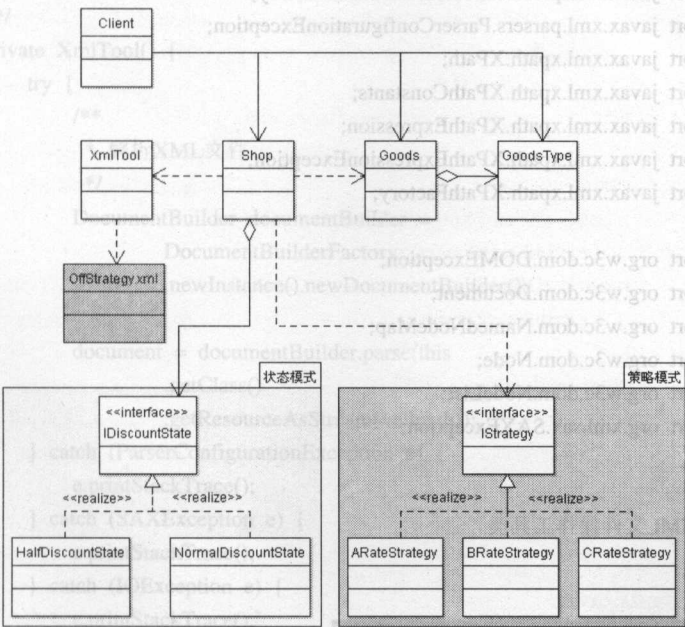


图23-6 超市打折综合解决方案类图

下面首先介绍的是策略的配置文件：

代码片段10 OffStrategy.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- 计算折扣的策略配置 -->
3 <strategies>
4
5     <strategy type="无分类"
6         class="cn.steven.pattern.demo.strategy.ARateStrategy" />
7
8     <strategy type="服装"
9         class="cn.steven.pattern.demo.strategy.BRateStrategy" />
10
11     <strategy type="食品"
12         class="cn.steven.pattern.demo.strategy.CRateStrategy" />
13
14 </strategies>

```

此文件的作用是采用外部配置的方式动态地确定某种商品具体使用的是哪种打折策略，这是一个格式良好的XML¹文件，XML文件是各种编程语言最常用的配置文件形式。

下面是操作XML的工具类，其中运用了dom²和XPath³的技术：

代码片段11 XmlTool.java

```

1 package cn.steven.pattern.demo.strategy;
2
3 import java.io.IOException;
4
5 import javax.xml.parsers.DocumentBuilder;
6 import javax.xml.parsers.DocumentBuilderFactory;
7 import javax.xml.parsers.ParserConfigurationException;
8 import javax.xml.xpath.XPath;
9 import javax.xml.xpath.XPathConstants;
10 import javax.xml.xpath.XPathExpression;
11 import javax.xml.xpath.XPathExpressionException;
12 import javax.xml.xpath.XPathFactory;
13
14 import org.w3c.dom.DOMException;
15 import org.w3c.dom.Document;
16 import org.w3c.dom.NamedNodeMap;
17 import org.w3c.dom.Node;
18 import org.w3c.dom.NodeList;
19 import org.xml.sax.SAXException;
20
21 /**
22  * XML文件操作工具类
23  */

```

¹<http://zh.wikipedia.org/wiki/XML>。

²<http://zh.wikipedia.org/zh-cn/%E6%96%87%E6%A1%A3%E5%AF%B9%E8%B1%A1%E6%A8%A1%E5%9E%8B>。

³<http://en.wikipedia.org/wiki/Xpath>。

```

24 public class XmlTool {
25
26     /**
27      * 配置文件路径
28      */
29     private String xmlpath
30         = "/cn/steven/pattern/demo/strategy/OffStrategy.xml";
31
32     /**
33      * dom文档模型
34      */
35     private Document document;
36
37     /**
38      * 采用 Initialization on Demand Holder idiom 构造单例类
39      * 内部静态类，单例持有者
40      */
41     static class SingletonHolder {
42         static XmlTool instance = new XmlTool();
43     }
44
45     public static XmlTool getInstance() {
46         return SingletonHolder.instance;
47     }
48
49     /**
50      * 私有构造方法
51      */
52     private XmlTool() {
53         try {
54             /**
55              * 解析XML文件
56              */
57             DocumentBuilder documentBuilder =
58                 DocumentBuilderFactory
59                     .newInstance().newDocumentBuilder();
60             document = documentBuilder.parse(this
61                 .getClass()
62                 .getResourceAsStream(xmlpath));
63         } catch (ParserConfigurationException e) {
64             e.printStackTrace();
65         } catch (SAXException e) {
66             e.printStackTrace();
67         } catch (IOException e) {
68             e.printStackTrace();
69         }
70     }
71
72     public String findStrategyClass(String key) {

```

```
75 if (document == null) {
76     return null;
77 }
78
79 try {
80     /**
81      * 使用XPath查找节点
82      */
83     XPathFactory xpathFactory = XPathFactory
84         .newInstance();
85     XPath xpath = xpathFactory.newXPath();
86
87     /**
88      * 构建一个XPath表达式
89      */
90     XPathExpression expression = xpath
91         .compile("/strategies/strategy[@type=\""
92             + key + "\"]");
93     // 根据XPathExpression对象查找对应的节点集合
94     NodeList nodes = (NodeList) expression
95         .evaluate(document,
96             XPathConstants.NODESET);
97     if (nodes != null) {
98         // 遍历节点集合
99         // 判断如果并未找出任何节点，则返回null
100         if (nodes.getLength() <= 0) {
101             return null;
102         }
103         Node node = nodes.item(0);
104         NamedNodeMap map = node.getAttributes();
105         // 返回属性值
106         return map.getNamedItem("class")
107             .getNodeValue();
108     }
109 } catch (XPathExpressionException e) {
110     e.printStackTrace();
111 } catch (DOMException e) {
112     e.printStackTrace();
113 }
114 return null;
115 }
116 }
117 }
```

商品类:

代码片段12 Goods.java

```
1 package cn.steven.pattern.demo.strategy;
2
3 /**
4  * 商品类
```



```

5  */
6  public class Goods {
7      /**
8       * 商品的类型
9       */
10     private GoodsType goodsType;
11
12     /**
13      * 商品价格
14      */
15     private float price;
16
17     /**
18      * 商品名称
19      */
20     private String name;
21
22     /**
23      * 构造方法
24      */
25     public Goods(GoodsType goodsType, String name,
26                 float price) {
27         super();
28         this.goodsType = goodsType;
29         this.price = price;
30         this.name = name;
31     }
32
33     public String getName() {
34         return name;
35     }
36
37     public void setName(String name) {
38         this.name = name;
39     }
40
41     public GoodsType getGoodsType() {
42         return goodsType;
43     }
44
45     public void setGoodsType(GoodsType goodsType) {
46         this.goodsType = goodsType;
47     }
48
49     public float getPrice() {
50         return price;
51     }
52
53     public void setPrice(float price) {
54         this.price = price;
55     }

```

56
57 }

商品类型:

代码片段13 GoodsType.java

```
1 package cn.steven.pattern.demo.strategy;  
2  
3 /**  
4  * 商品类型枚举  
5  */  
6 public enum GoodsType {  
7     无分类,  
8     服装,  
9     食品  
10 }
```

状态接口:

代码片段14 IDiscountState.java

```
1 package cn.steven.pattern.demo.strategy;  
2  
3 /**  
4  * 折扣状态接口  
5  */  
6 public interface IDiscountState {  
7     /**  
8      * 价格系数，如全价为1，半价为0.5  
9      *  
10     * @return  
11     */  
12     float getDiscount();  
13  
14     /**  
15      * 文字描述  
16      *  
17     * @return  
18     */  
19     String getMsg();  
20 }
```

全价状态:

代码片段15 NormalDiscountState.java

```
1 package cn.steven.pattern.demo.strategy;  
2  
3 /**  
4  * 不打折扣  
5  */  
6 public class NormalDiscountState implements IDiscountState {  
7
```

```

8      /**
9       * 价格系数
10     */
11     private float discount = 1;
12
13     public float getDiscount() {
14         return discount;
15     }
16
17     public void setDiscount(float discount) {
18         this.discount = discount;
19     }
20
21     /**
22     * 状态描述
23     */
24     private String msg = "没有折扣";
25
26     public String getMsg() {
27         return msg;
28     }
29
30     public void setMsg(String msg) {
31         this.msg = msg;
32     }
33
34     /**
35     * 采用 Initialization on Demand Holder idiom 构造单例类
36     *
37     * 内部静态类，单例持有者
38     */
39     static class SingletonHolder {
40         static NormalDiscountState instance
41             = new NormalDiscountState();
42     }
43
44     public static NormalDiscountState getInstance() {
45         return SingletonHolder.instance;
46     }
47
48     private NormalDiscountState() {
49     }
50 }
51
52 }

```

半价状态:

代码片段16 HalfDiscountState.java

```

1 package cn.steven.pattern.demo.strategy;
2

```



```

3  /**
4   * 半价折扣
5   */
6  public class HalfDiscountState implements IDiscountState {
7
8      /**
9       * 折扣
10      */
11     private float discount = 0.5f;
12
13     public float getDiscount() {
14         return discount;
15     }
16
17     public void setDiscount(float discount) {
18         this.discount = discount;
19     }
20
21     /**
22      * 状态描述
23      */
24     private String msg = "半价";
25
26     public String getMsg() {
27         return msg;
28     }
29
30     public void setMsg(String msg) {
31         this.msg = msg;
32     }
33
34     /**
35      * 采用 Initialization on Demand Holder idiom 构造单例类
36      *
37      * 内部静态类，单例持有者
38      */
39     static class SingletonHolder {
40         static HalfDiscountState instance = new HalfDiscountState();
41     }
42
43     public static HalfDiscountState getInstance() {
44         return SingletonHolder.instance;
45     }
46
47     private HalfDiscountState() {
48
49     }
50
51 }

```

商品打折策略接口:

代码片段17 IStrategy.java

```

1 package cn.steven.pattern.demo.strategy;
2
3 /**
4  * 打折策略
5  */
6 public interface IStrategy {
7     /**
8      * 获得折扣比例
9      *
10     * @return
11     */
12     public float getRate();
13 }

```

策略A:

代码片段18 ARateStrategy.java

```

1 package cn.steven.pattern.demo.strategy;
2
3 /**
4  * 全价策略
5  */
6 public class ARateStrategy implements IStrategy {
7
8     /**
9      * 获得折扣比例
10     *
11     * @return
12     */
13     @Override
14     public float getRate() {
15         // TODO Auto-generated method stub
16         return 1;
17     }
18
19 }

```

策略B:

代码片段19 BRateStrategy.java

```

1 package cn.steven.pattern.demo.strategy;
2
3 /**
4  * 8折策略
5  */
6 public class BRateStrategy implements IStrategy {
7
8     /**
9      * 获得折扣比例
10     *

```

```
11  * @return
12  */
13  @Override
14  public float getRate() {
15      // TODO Auto-generated method stub
16      return .8f;
17  }
18
19 }
```

策略C:

代码片段20 CRateStrategy.java

```
1  package cn.steven.pattern.demo.strategy;
2
3  /**
4   * 5折策略
5   */
6  public class CRateStrategy implements IStrategy {
7
8      /**
9       * 获得折扣比例
10      *
11      * @return
12      */
13      @Override
14      public float getRate() {
15          // TODO Auto-generated method stub
16          return .5f;
17      }
18
19 }
```

商店类:

代码片段21 Shop.java

```
1  package cn.steven.pattern.demo.strategy;
2
3  /**
4   * 商店单例类
5   * 计算折扣时先计算状态折扣，再计算商品类型折扣
6   */
7  public class Shop {
8
9      /**
10       * 聚合状态对象
11       */
12      private IDiscountState discountState;
13
14      public IDiscountState getDiscountState() {
15          return discountState;
```



```

16     }
17
18     public void setDiscountState(
19         IDiscountState discountState) {
20         this.discountState = discountState;
21     }
22
23     /**
24      * 聚合策略对象
25      */
26     private IStrategy strategy;
27
28     /**
29      * 计算折扣后的金额
30      */
31     public double check(Goods goods) {
32         /**
33          * 计算状态折扣
34          */
35         float money = goods.getPrice()
36             * discountState.getDiscount();
37
38         try {
39             /**
40              * 计算策略折扣
41              */
42             strategy = (IStrategy) Class.forName(
43                 XmlTool.getInstance()
44                     .findStrategyClass(
45                         goods.getGoodsType()
46                             .name())
47                     .newInstance());
48             money *= strategy.getRate();
49         } catch (Exception e) {
50             e.printStackTrace();
51         }
52
53         System.out.println(goods.getName() + "["
54             + goods.getGoodsType() + "]" + " 原价:"
55             + goods.getPrice() + " 全场折扣:"
56             + discountState.getDiscount() + " 商品优惠:"
57             + strategy.getRate() + " 最终价格:" + money);
58         return money;
59     }
60
61     /**
62      * 采用 Initialization on Demand Holder idiom 构造单例类
63      *
64      * 内部静态类，单例持有者
65      */
66     static class SingletonHolder {

```

```
67         static Shop instance = new Shop();
68     }
69     @Override
70     public static Shop getInstance() {}
71     return SingletonHolder.instance;
72 }
73
74 private Shop() {
75 }
76 }
77
78 }
```

客户端代码:

代码片段22 Client.java

```
1 package cn.steven.pattern.demo.strategy;
2
3 /**
4  * 策略模式+状态模式
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9         /**
10          * 创建环境
11          */
12         Shop shop = Shop.getInstance();
13
14         /**
15          * 设置当前状态
16          */
17         shop.setDiscountState(HalfDiscountState
18             .getInstance());
19
20         /**
21          * 创建几个商品
22          */
23         Goods g1 = new Goods(GoodsType.服装, "牛仔褲", 55.8f);
24         Goods g2 = new Goods(GoodsType.食品, "牛肉", 9.8f);
25         Goods g3 = new Goods(GoodsType.无分类, "平底锅", 120.6f);
26
27         /**
28          * Shop {
29          * 计算价格
30          */
31         shop.check(g1);
32         shop.check(g2);
33         shop.check(g3);
34
35         /**
36          * IDiscountState
37          * 改变状态
38          */
39     }
```

```
36      */
37      shop.setDiscountState(HalfDiscountState
38          .getInstance());
39      /**
40       * 计算价格
41       */
42      shop.check(g1);
43      shop.check(g2);
44      shop.check(g3);
45  }
46 }
```

运行结果如图23-7所示。

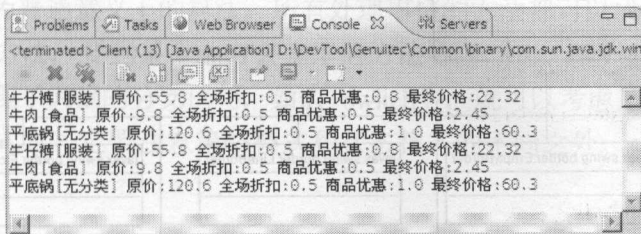


图23-7 Client.java运行结果

由客户端代码和图23-7可见，通过状态模式和策略模式的结合使用，现在已经可以成功地控制一个商品的打折过程了，并且如果想添加一个状态或一个策略也非常容易，客户端的代码也变得非常稳定。

23.2.3 策略模式在JDK中的实例

在JDK中可以找到很多实际的策略模式的例子，比如在javax.swing包中可以看到，很多组件都需要画出边框，比如JPanel，JButton等组件，可以选择的边框类型也有很多，如CompoundBorder、EmptyBorder、EtchedBorder、LineBorder等。

在面临这么多的边框架构时应该如何选择呢？当一件事情可以有多种做法的时候，就可以考虑使用策略模式了。

下面看一下Java是如何处理边框接口的：

public interface Border

该接口描述一个能够围绕swing组件边缘的边框对象。有关使用Border的示例，请参阅The Java Tutorial中的How to Use Borders一节。

在Swing组件集中，Border取代了Insets作为一种创建组件边缘四周的装饰或普通区域的机制。

用法说明：

- 使用EmptyBorder创建普通边框（该机制取代了原先的setInsets）。
- 使用嵌套多个Border对象的CompoundBorder来创建单个组合边框。
- Border实例设计为可共享。不使用某个Border类来创建新的Border对象，而是使用BorderFactory方法生成常见Border类型的共享实例。
- 其他边框样式包括BevelBorder、SoftBevelBorder、EtchedBorder、LineBorder、

TitledBorder和MatteBorder。

- 要创建新的Border类，请用AbstractBorder创建子类。

其类层次如图23-8所示。

由图23-8可见通过这样的一个设计，JComponent的子类就可以通过更改自身的边框策略来改变样式了，且增加一个新的边框样式也非常简单。

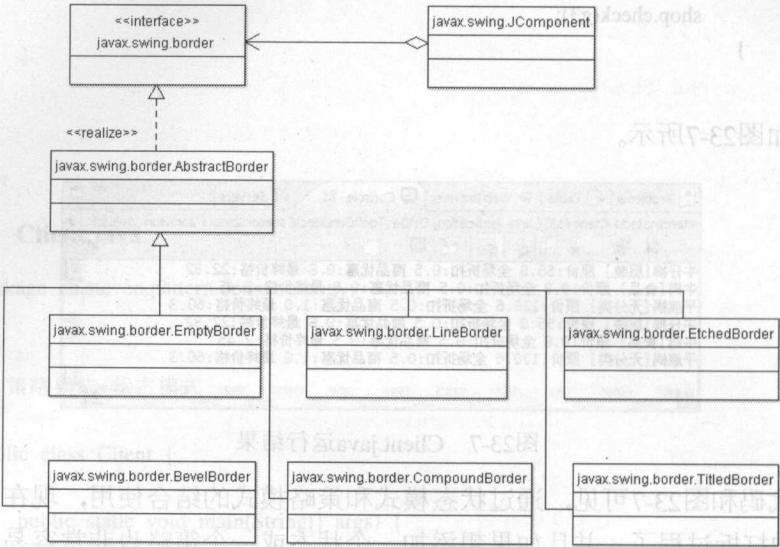


图23-8 JDK边框策略类图

23.2.4 策略模式的使用范围及优点

策略模式的优点：

- 简化了单元测试，因为每个算法都有自己的类，可以通过自己的接口单独进行测试。
- 避免程序中使用多重条件转移语句，以便使系统更灵活，并易于扩展。
- 遵守大部分GRASP原则和常用设计原则，高内聚、低耦合。

策略模式的缺点：

- 因为每个具体策略类都会产生一个新类，所以会增加系统需要维护的类的数量。
- 在基本的策略模式中，选择所用的具体实现的职责由客户端对象承担，并转给策略模式的Context对象（这本身没有解除客户端需要选择判断的压力，而策略模式与简单工厂模式结合后，选择具体实现的职责也可以由Context来承担，这就最大化地减轻了客户端的压力）。

策略模式的应用范围：

- 许多相关的类仅仅是行为有异的情况。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。
- 需要使用一个算法的不同变体的情况。例如，你可能会定义一些反映不同的空间/时间权衡的算法，当这些变体实现为一个算法的类层次时，可以使用策略模式。
- 要避免暴露复杂的、与算法相关的数据结构的情况。
- 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现的情况。这时可将相关的条件分支移入它们各自的策略类中以代替这些条件语句。

又不允许进行销售或进货操作。其规则列表如下。

23.2.5 与其他模式的关系

策略模式在结构上与状态模式非常相似，但是它们的目的性差异非常大。区分这两个模式的关键是行为是由状态驱动还是由一组算法驱动。通常，状态模式的“状态”是在对象内部的，策略模式的“策略”是外部赋予的，就关系而言，状态与对象的关系更加紧密一些。

23.3 策略模式总结

策略模式可以定义一系列算法，把它们一个个封装起来，并且使它们可以相互替换。该模式使得算法可独立于它们的客户端而变化。注意这里讲到了一个算法问题，读者在理解定义的时候不要将算法理解为普通意义上的算法。所有处理事情的方法都可以成为算法，比如打折的方法，格式化文本的方法，要到某个地方去坐车的方法等。

如果在程序中出现了大量的分支语句，如if、switch等，就可以考虑是否可以用策略模式来进行重构。在选择模式时一定要注意策略模式和状态模式的不同之处。

图23-3 策略模式结构图

图23-3展示了策略模式的结构。它包含一个抽象策略接口（Strategy），一个或多个具体策略实现（ConcreteStrategy），以及一个上下文（Context）。Context接口定义了一个方法，该方法调用具体策略实现的方法。ConcreteStrategy实现类实现了Context接口定义的方法。图23-3还展示了策略模式的代码实现。图23-3的代码实现如下：

```
1 package org.steven.pattern.demo.mediator.queue;
2
3 /**
4  * 策略
5  */
6 public class Strategy {
7
8     // 策略
9
10    // 策略
11    private int count;
12
13    public int getCount() {
14        return count;
15    }
16
17    public void setCount(int count) {
18        this.count = count;
19    }
20
21 }
```

图23-3 策略模式结构图

第24章 调停者模式 (Mediator)

调停者模式定义了一个可以封装一组对象之间的相互影响的行为的对象，这样可以使用松耦合的方式联系一组对象，避免对象之间互相显式地直接引用，从而当改变了某些对象之间的关系时，就可以不影响其他的对象了¹。

当系统的参与者有很多时，通常这些对象之间会产生各种关系，这就像在工作中，各个同事之间的关系一样。假设有一个六人组成的团队，其关系可能如图24-1所示。

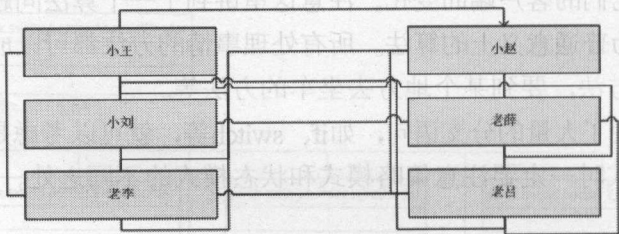


图24-1 团队成员关系

由图24-1可见团队成员中的关系错综复杂，每一个同事都可能和其他同事联系，任何一种关系的改变都可能影响到其他同事，因此改变团队的结构也变得极为困难。

根据管理学的经验，可以引入一个经理的职位，此职位的主要功能就是管理各个同事之间的关系，使得各个同事在做事情的时候无需和其他同事直接打交道，先和经理互动即可。这样如果某个同事的职责有改变，其他同事也无需关心，因为各个同事之间的耦合度是非常低的，可以相互独立地进行变化，如图24-2所示。

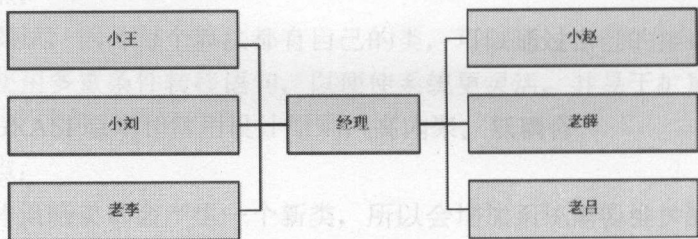


图24-2 团队成员关系改进

由图24-2可见，引入了经理职位后，各个同事间的关系变得非常简洁了。这种设计方式正是调停者模式的设计理念。

24.1 超市经营的核心——库存管理

超市经营中有一个最为重要的问题就是：库存管理。超市中的很多操作都与库存有关，其中很多操作对于其他操作又是有排斥作用的，比如在销售和进货时不允许进行盘点，而盘点时

¹Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.GOF[95]

又不允许进行销售或进货操作。其规则列表如下：

- 销售时：可以进货，不许盘点。
- 进货时：可以销售，不许盘点。
- 盘点时：不许销售，不许进货。

为了简化设计，可以不操作不受影响的对象，比如销售时允许进货，不允许盘点，则只操作不允许的对象即可。初步的设计会把相关的对象耦合在一起，类图如图24-3所示。

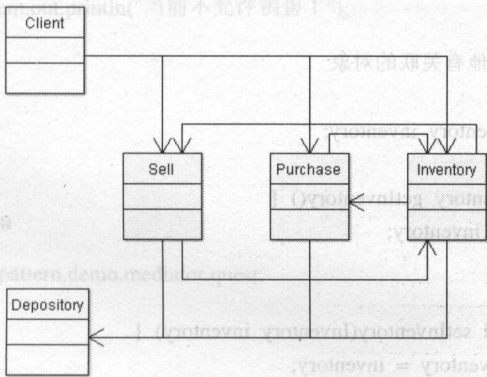


图24-3 初步设计类图

由图24-3可见，类之间的关系错综复杂，当相互作用的类增多时，关系呈几何级增长。具体的代码如下所示。

仓库类，对于此类我们只设计了简单的属性：

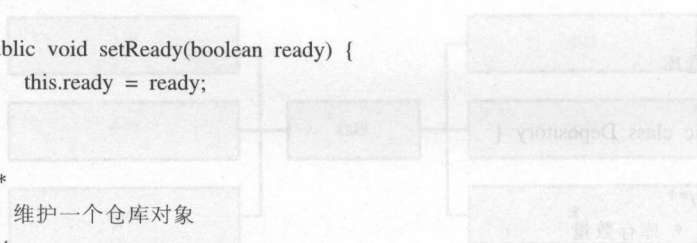
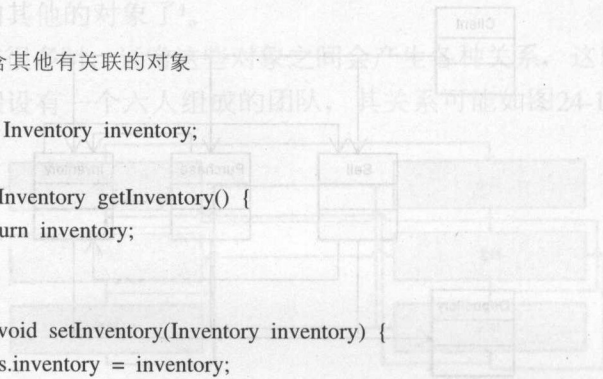
代码片段1 Depository.java

```
1 package cn.steven.pattern.demo.mediator.quest;
2
3 /**
4  * 仓库
5  */
6 public class Depository {
7
8     /**
9      * 库存数量
10     */
11     private int count;
12
13     public int getCount() {
14         return count;
15     }
16
17     public void setCount(int count) {
18         this.count = count;
19     }
20
21 }
```

销售类：

代码片段2 Sell.java

```
1 package cn.steven.pattern.demo.mediator.quest;
2
3 /**
4  * 销售
5  */
6 public class Sell {
7
8     /**
9      * 包含其他有关联的对象
10     */
11     private Inventory inventory;
12
13     public Inventory getInventory() {
14         return inventory;
15     }
16
17     public void setInventory(Inventory inventory) {
18         this.inventory = inventory;
19     }
20
21     /**
22      * 是否允许操作
23      */
24     private boolean ready = true;
25
26     public boolean isReady() {
27         return ready;
28     }
29
30     public void setReady(boolean ready) {
31         this.ready = ready;
32     }
33
34     /**
35      * 维护一个仓库对象
36      */
37     private Depository depository;
38
39     public Depository getDepository() {
40         return depository;
41     }
42
43     public void setDepository(Depository depository) {
44         this.depository = depository;
45     }
46
47     /**
48      * 工作方法
49      */
50     public void doSell() {
```



24.1 超市管理

超市管理 一个最为重要的问题就是：库存管理。超市中的很多操作都与库存有关，其对于其他操作又有排斥作用的，比如在销售和进货时不允许进行盘点，而盘点时

```

51         if (ready) {
52             System.out.println("=====销售=====");
53             System.out.println("准备销售，不允许盘点！");
54             inventory.setReady(false);
55             System.out.println("销售.....done");
56             System.out.println("恢复盘点！");
57             inventory.setReady(true);
58         } else {
59             System.out.println("当前不允许销售！");
60         }
61     }
62 }

```

进货类:

代码片段3 Purchase.java

```

1  package cn.steven.pattern.demo.mediator.quest;
2
3  /**
4   * 采购进货
5   */
6  public class Purchase {
7
8      /**
9       * 包含其他有关联的对象
10      */
11     private Inventory inventory;
12
13     public Inventory getInventory() {
14         return inventory;
15     }
16
17     public void setInventory(Inventory inventory) {
18         this.inventory = inventory;
19     }
20
21     /**
22      * 是否允许操作
23      */
24     private boolean ready = true;
25
26     public boolean isReady() {
27         return ready;
28     }
29
30     public void setReady(boolean ready) {
31         this.ready = ready;
32     }
33
34     /**
35      * 维护一个仓库对象

```



```

36     */
37     private Depository depository;
38
39     public Depository getDepository() {
40         return depository;
41     }
42
43     public void setDepository(Depository depository) {
44         this.depository = depository;
45     }
46
47     /**
48      * 工作方法
49      */
50     public void doPurchase() {
51         if (ready) {
52             System.out.println("=====进货=====");
53             System.out.println("准备进货，不允许盘点！");
54             inventory.setReady(false);
55             System.out.println("进货.....done");
56             System.out.println("恢复盘点！");
57             inventory.setReady(true);
58         } else {
59             System.out.println("当前不允许进货！");
60         }
61     }
62 }

```

盘点类:

代码片段4 Inventory.java

```

1 package cn.steven.pattern.demo.mediator.quest;
2
3 /**
4  * 盘点
5  */
6 public class Inventory {
7
8     /**
9      * 包含其他有关联的对象
10     */
11     private Purchase purchase;
12
13     private Sell sell;
14
15     public Purchase getPurchase() {
16         return purchase;
17     }
18
19     public void setPurchase(Purchase purchase) {
20         this.purchase = purchase;

```

```
21     }
22
23     public Sell getSell() {
24         return sell;
25     }
26
27     public void setSell(Sell sell) {
28         this.sell = sell;
29     }
30
31     /**
32      * 是否允许操作
33      */
34     private boolean ready = true;
35
36     public boolean isReady() {
37         return ready;
38     }
39
40     public void setReady(boolean ready) {
41         this.ready = ready;
42     }
43
44     /**
45      * 维护一个仓库对象
46      */
47     private Depository depository;
48
49     public Depository getDepository() {
50         return depository;
51     }
52
53     public void setDepository(Depository depository) {
54         this.depository = depository;
55     }
56
57     /**
58      * 工作方法
59      */
60     public void doInventory() {
61         if (ready) {
62             System.out.println("=====盘点=====");
63             System.out.println("准备盘点, 不允许进货!");
64             purchase.setReady(false);
65             System.out.println("准备盘点, 不允许销售!");
66             sell.setReady(false);
67             System.out.println("仓库盘点.....done");
68             System.out.println("恢复进货!");
69             purchase.setReady(true);
70             System.out.println("恢复销售!");
71             sell.setReady(true);
```

```
72     } else {
73         private System.out.println("当前不允许盘点！");
74     }
75 } public Depository getDepository() {
76     return depository;
```

客户端类:

代码片段5 Client.java

```
1 package cn.steven.pattern.demo.mediator.quest;
2
3 /**
4  * 客户端
5  */
6 public class Client {
7     public static void main(String[] args) {
8
9         /**
10          * 创建需要操作的对象
11          */
12         Depository depository = new Depository();
13         Sell sell = new Sell();
14         Purchase purchase = new Purchase();
15         Inventory inventory = new Inventory();
16
17         /**
18          * 组装关联的对象
19          */
20         sell.setDepository(depository);
21         sell.setInventory(inventory);
22
23         purchase.setDepository(depository);
24         purchase.setInventory(inventory);
25
26         inventory.setDepository(depository);
27         inventory.setSell(sell);
28         inventory.setPurchase(purchase);
29
30         /**
31          * 运行
32          */
33         purchase.doPurchase();
34         sell.doSell();
35         inventory.doInventory();
36
37     }
38
39 }
```

运行结果如图24-4所示。

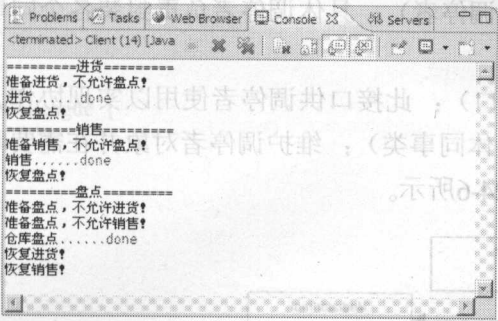


图24-4 客户端运行结果

由以上运行结果和代码可见，结果是正确的，一个对象在操作时它会操作与之相关联的对象。但是经过思考，就可以发现这样的设计存在如下问题：

- 关联对象之间的耦合太过紧密。
- 更改一个对象时，将会影响到所有与之相关联的类。
- 结构不易扩充，不稳定。
- 违反了“最少知识原则”。

一组相互关联的类通常被称为“同事类”，当同事类过多时，使用调停者模式可以很好地组织其之间的关系。

24.2 调停者模式的结构

调停者模式将相互关联的一系列对象封装了起来，它通常使用的是聚合的方式，每一个被聚合的对象的内部同时也聚合了一个调停者对象，这样调停者对象和各个同事对象就可以相互调用，但是各个同事之间是不允许相互访问的。

24.2.1 调停者模式

调停者模式设计的原始类图如图24-5所示。

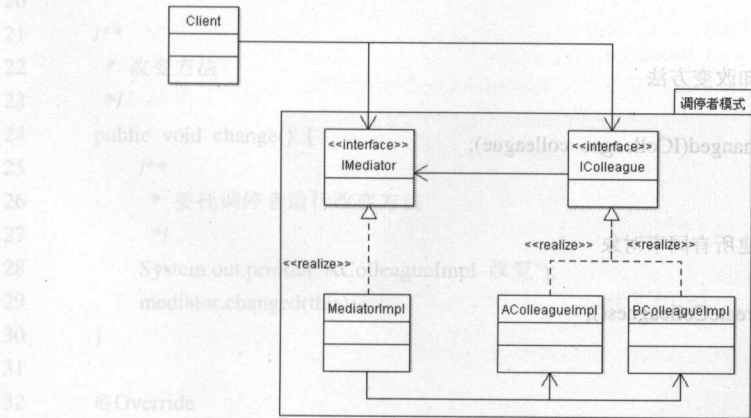


图24-5 调停者模式类图

图24-5中的参与者如下：

- IMediator（调停者接口）：此接口用于与各个同事（IColleague）对象通信。

- MediatorImpl（具体调停者）：具体调停者负责封装各个同事对象之间的协作关系并维护各个同事对象的实例。
 - IColleague（同事接口）：此接口供调停者使用以实现协作。
 - XColleagueImpl（具体同事类）：维护调停者对象并在需要时调用调停者的协作方法。
- 该模式的序列图如图24-6所示。

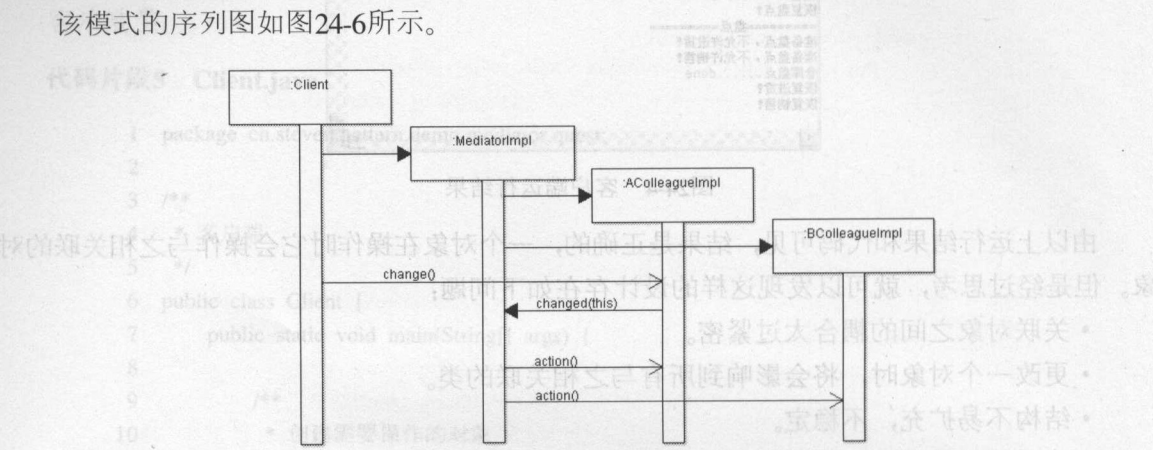


图24-6 调停者模式序列图

由图24-6可见，当某一个同事类发生改变时，它将会通知调停者对象，由调停者对象接下来通知各个相关的同事对象，这样就避免了同事对象之间的相互耦合。

下面展示的是具体的实现代码。
调停者接口：

代码片段6 IMediator.java

```
1 package cn.steven.pattern.demo.mediator.pattern;
2
3 /**
4  * 调停者接口
5  */
6 public interface IMediator {
7
8     /**
9      * 通知改变方法
10     */
11     void changed(IColleague colleague);
12
13     /**
14      * 创建所有同事对象
15     */
16     void createColleagues();
17 }
```

同事接口：

代码片段7 IColleague.java

```
1 package cn.steven.pattern.demo.mediator.pattern;
2
```

```

3  /**
4   * 同事接口
5   */
6  public interface IColleague {
7
8      /**
9       * 工作方法
10      */
11      void action();
12
13  }

```

同事实现类A:

代码片段8 AColleagueImpl.java

```

1  package cn.steven.pattern.demo.mediator.pattern;
2
3  /**
4   * 同事A
5   */
6  public class AColleagueImpl implements IColleague {
7
8      /**
9       * 聚合调停者对象
10     */
11     private IMediator mediator;
12
13     public IMediator getMediator() {
14         return mediator;
15     }
16
17     public void setMediator(IMediator mediator) {
18         this.mediator = mediator;
19     }
20
21     /**
22      * 改变方法
23      */
24     public void change() {
25         /**
26          * 委托调停者运行改变方法
27          */
28         System.out.println("AColleagueImpl 改变");
29         mediator.changed(this);
30     }
31
32     @Override
33     public void action() {
34         System.out.println("AColleagueImpl 得到运行");
35     }
36
37 }

```


同事实现类B:

代码片段9 BColleagueImpl.java

```
1 package cn.steven.pattern.demo.mediator.pattern;
2
3 /**
4  * 同事B
5  */
6 public class BColleagueImpl implements IColleague {
7
8     /**
9      * 聚合调停者对象
10     */
11     private IMediator mediator;
12
13     public IMediator getMediator() {
14         return mediator;
15     }
16
17     public void setMediator(IMediator mediator) {
18         this.mediator = mediator;
19     }
20
21     /**
22      * 改变方法
23     */
24     public void change() {
25         /**
26          * 委托调停者运行改变方法
27          */
28         System.out.println("BColleagueImpl 改变");
29         mediator.changed(this);
30     }
31
32     @Override
33     public void action() {
34         System.out.println("BColleagueImpl 得到运行");
35     }
36
37 }
```

调停者实现类:

代码片段10 MediatorImpl.java

```
1 package cn.steven.pattern.demo.mediator.pattern;
2
3 /**
4  * 具体调停者
5  */
6 public class MediatorImpl implements IMediator {
7
```

```

8      /**
9      * 聚合同事对象
10     */
11     AColleagueImpl aColleagueImpl;
12     BColleagueImpl bColleagueImpl;
13
14     public AColleagueImpl getAColleagueImpl() {
15         return aColleagueImpl;
16     }
17
18     public void setAColleagueImpl(AColleagueImpl colleagueImpl) {
19         aColleagueImpl = colleagueImpl;
20     }
21
22     public BColleagueImpl getBColleagueImpl() {
23         return bColleagueImpl;
24     }
25
26     public void setBColleagueImpl(BColleagueImpl colleagueImpl) {
27         bColleagueImpl = colleagueImpl;
28     }
29
30     /**
31     * 通知改变方法
32     */
33     @Override
34     public void changed(IColleague colleague) {
35         /**
36         * 标记改变时的处理
37         */
38         aColleagueImpl.action();
39         bColleagueImpl.action();
40     }
41
42     /**
43     * 创建所有同事对象
44     */
45     @Override
46     public void createColleagues() {
47
48         /**
49         * 实例化
50         */
51         aColleagueImpl = new AColleagueImpl();
52         bColleagueImpl = new BColleagueImpl();
53
54         /**
55         * 设置依赖
56         */
57         aColleagueImpl.setMediator(this);
58         bColleagueImpl.setMediator(this);

```

```
59 }
60
61 }
```

客户端代码:

代码片段11 Client.java

```
1 package cn.steven.pattern.demo.mediator.pattern;
2
3 /**
4  * 客户端
5  */
6 public class Client {
7     public static void main(String[] args) {
8
9         /**
10          * 创建调停者
11          */
12         MediatorImpl mediator = new MediatorImpl();
13
14         /**
15          * 构造所有同事对象
16          */
17         mediator.createColleagues();
18
19         /**
20          * 获得同事对象
21          */
22         AColleagueImpl aColleagueImpl = mediator.getAColleagueImpl();
23         BColleagueImpl bColleagueImpl = mediator.getBColleagueImpl();
24
25         /**
26          * 对象互动
27          */
28         aColleagueImpl.change();
29         System.out.println();
30         bColleagueImpl.change();
31
32     }
33 }
```

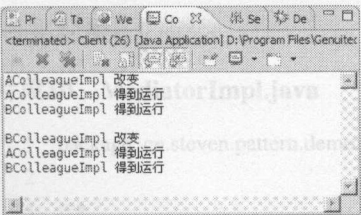


图24-7 运行结果

代码运行结果如图24-7所示。

由图24-7可见，通过使用调停者模式，各个同事类可以独立操作了，而且将同事间的关系交由调停者对象处理，也使得同事类之间的耦合关系被解开了，并且各个同事类可以独立变化。

24.2.2 用调停者模式协调进货/销售/盘点之间的关系

经过了上一节的学习，读者已经可以初步掌握调停者模式的设计方案了，但是如果要将此模式应用于实际需求，通常还需要做一些更改。

由前面一节可知，一个好的库存管理是很复杂的，影响到库存管理的外部同事对象也很多，如何将其解耦合呢？使用调停者模式时如何令调停者对象在同事类发生改变时自动得知呢？

考虑到以上问题，可以得出一个架构：可以结合使用观察者模式和调停者模式解决同事对象解耦合和自动发现的问题。

解决方案的类图如图24-8所示。

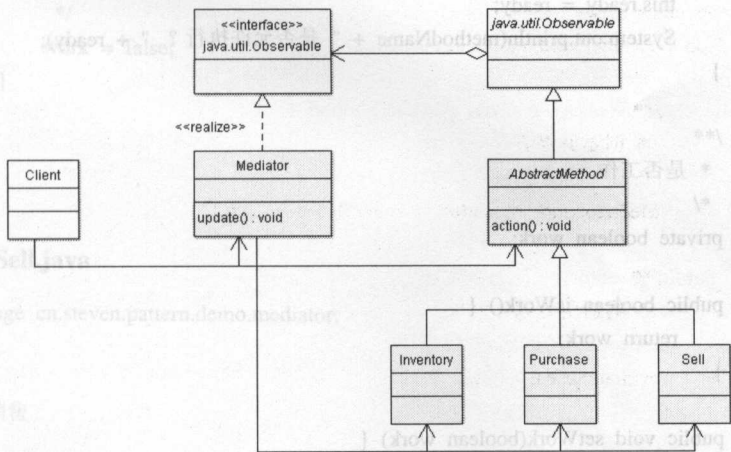


图24-8 结合使用调停者模式与观察者模式的解决方案类图

下面展示实现代码，首先是同事类的抽象：

代码片段12 AbstractMethod.java

```
1 package cn.steven.pattern.demo.mediator;
2
3 import java.util.Observable;
4
5 /**
6  * 抽象的超市操作方法
7  */
8 public abstract class AbstractMethod extends Observable {
9
10     /**
11      * 操作的名称
12      */
13     private String methodName;
14
15     public String getMethodName() {
16         return methodName;
17     }
18
19     public void setMethodName(String methodName) {
20         this.methodName = methodName;
```

```

21     }
22
23     /**
24      * 是否允许操作
25      */
26     private boolean ready;
27
28     public boolean isReady() {
29         return ready;
30     }
31
32     public void setReady(boolean ready) {
33         this.ready = ready;
34         System.out.println(methodName + " 是否允许执行？" + ready);
35     }
36
37     /**
38      * 是否工作
39      */
40     private boolean work;
41
42     public boolean isWork() {
43         return work;
44     }
45
46     public void setWork(boolean work) {
47         this.work = work;
48
49         /**
50          * 通知观察者已经改变
51          */
52         this.setChanged();
53         this.notifyObservers();
54     }
55
56     /**
57      * 工作方法
58      */
59     public void action() {
60         if (ready) {
61             System.out.println("===== " + methodName + " =====");
62             /**
63              * 工作过程
64              */
65             this.setWork(true);
66             System.out.println(methodName + " .....done !");
67             this.setWork(false);
68         } else {
69             System.out.println("当前不允许" + methodName + " !");
70         }
71     }

```

```
72 public void setInventory(Inventory inventory) { }
73 /** this.inventory = inventory;
74 * 构造方法
75 */
76 public AbstractMethod() {
77     /**
78     * 设置就绪
79     */
80     ready = true;
81
82     /**
83     * 设置未工作
84     */
85     work = false;
86 }
87
88 }
```

销售类:

代码片段13 Sell.java

```
1 package cn.steven.pattern.demo.mediator;
2
3 /**
4 * 销售
5 */
6 public class Sell extends AbstractMethod {
7
8     /** inventory = new Inventory();
9     * 构造方法
10    */
11    public Sell() {
12        super();
13        this.setMethodName("销售");
14    }
15
16 }
```

进货类:

代码片段14 Purchase.java

```
1 package cn.steven.pattern.demo.mediator;
2
3 /**
4 * 采购进货
5 */
6 public class Purchase extends AbstractMethod {
7
8     /**
9     * 构造方法
10    */
```



```

11     public Purchase() {
12         super();
13         this.setMethodName("采购进货");
14     }
15
16 }

```

盘点类:

代码片段15 Inventory.java

```

1  package cn.steven.pattern.demo.mediator;
2
3  /**
4   * 盘点
5   */
6  public class Inventory extends AbstractMethod {
7
8      /**
9       * 构造方法
10      */
11     public Inventory() {
12         super();
13         this.setMethodName("盘点");
14     }
15
16 }

```

调停者类:

代码片段16 Mediator.java

```

1  package cn.steven.pattern.demo.mediator;
2
3  import java.util.Observable;
4  import java.util.Observer;
5
6  /**
7   * 调停者类
8   */
9  public class Mediator implements Observer {
10
11      /**
12       * 聚合同事对象
13       */
14     private Inventory inventory;
15     private Purchase purchase;
16     private Sell sell;
17
18     public Inventory getInventory() {
19         return inventory;
20     }
21

```

```
22 public void setInventory(Inventory inventory) {
23     this.inventory = inventory;
24 }
25
26 public Purchase getPurchase() {
27     return purchase;
28 }
29
30 public void setPurchase(Purchase purchase) {
31     this.purchase = purchase;
32 }
33
34 public Sell getSell() {
35     return sell;
36 }
37
38 public void setSell(Sell sell) {
39     this.sell = sell;
40 }
41
42 /**
43  * 创建所有同事对象
44  */
45 public void createMethods() {
46
47     /**
48      * 创建
49      */
50     inventory = new Inventory();
51     purchase = new Purchase();
52     sell = new Sell();
53
54     /**
55      * 关联观察者对象
56      */
57     inventory.addObserver(this);
58     purchase.addObserver(this);
59     sell.addObserver(this);
60 }
61
62 /**
63  * 观察者更新方法
64  */
65 @Override
66 public void update(Observable o, Object arg) {
67
68     /**
69      * 转换为同事对象
70      */
71     AbstractMethod abstractMethod = (AbstractMethod) o;
72 }
```

```

73      /**
74       * 业务逻辑
75       */
76       if (abstractMethod instanceof Sell) {
77           // 销售时：可以进货，不许盘点
78           if (abstractMethod.isWork()) {
79               inventory.setReady(false);
80           } else {
81               inventory.setReady(true);
82           }
83       } else if (abstractMethod instanceof Purchase) {
84           // 进货时：可以销售，不许盘点
85           if (abstractMethod.isWork()) {
86               inventory.setReady(false);
87           } else {
88               inventory.setReady(true);
89           }
90       } else if (abstractMethod instanceof Inventory) {
91           // 盘点时：不许销售，不许进货
92           if (abstractMethod.isWork()) {
93               sell.setReady(false);
94               purchase.setReady(false);
95           } else {
96               sell.setReady(true);
97               purchase.setReady(true);
98           }
99       }
100     }
101
102 }

```

客户端代码：

代码片段17 Client.java

```

1  package cn.steven.pattern.demo.mediator;
2
3  /**
4   * 客户端
5   */
6  public class Client {
7
8      /**
9       *
10      */
11      * 创建调停者
12      */
13      Mediator mediator = new Mediator();
14
15      /**
16      * 创建所有同事对象并关联观察者对象
17      */

```



```
18 mediator.createMethods();
19
20 /**
21  * 获得对象
22  */
23 Inventory inventory = mediator.getInventory();
24 Purchase purchase = mediator.getPurchase();
25 Sell sell = mediator.getSell();
26
27 /**
28  * 操作
29  */
30 sell.action();
31 purchase.action();
32 inventory.action();
33 }
34
35 }
```

代码的运行结果如图24-9所示。

结合代码和结果图可见，使用调停者模式的确可以达到预期的运行效果，且和观察者模式协作后将会更有效地简化程序员的编码工作。

24.2.3 调停者模式在JDK中的实例

调停者模式在JDK中并不多见，但是在真正的系统设计中是经常见到的，典型的应用是一个GUI用户的界面，如图24-10所示。

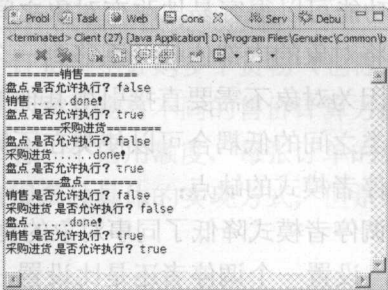


图24-9 代码运行结果

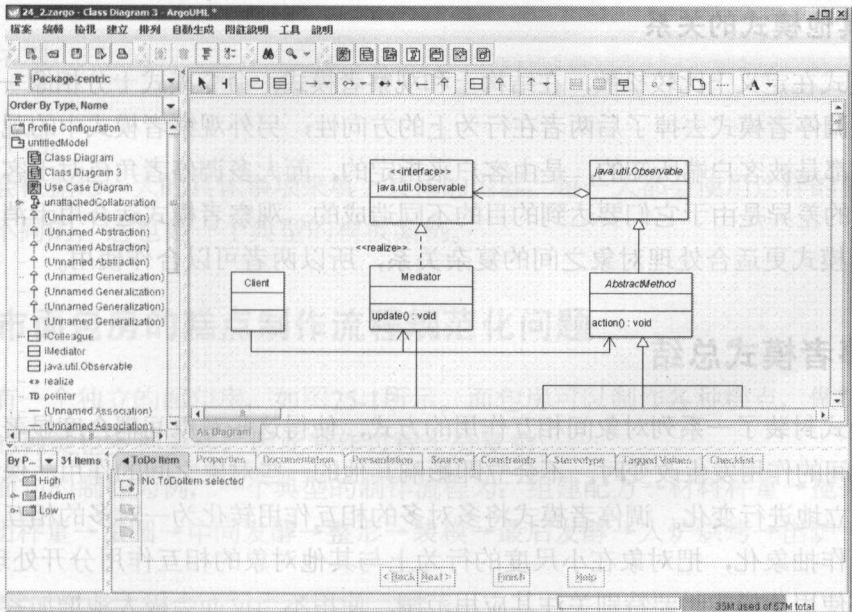


图24-10 一个GUI界面的示例

在这样一个复杂的用户界面中，各个组件（按钮、工具栏、选项、菜单）之间都可能存在千丝万缕的联系，比如在选择某种工具的时候某种操作不可以做，但是选择其他工具时这种操作是可以做的。

在网站设计中也经常遇到类似的情况，比如在未正确填写各种信息时，“注册”按钮是不可用的，所以在实际的Java编程过程中，很多地方都会用到调停者模式。读者应仔细思考其应用的情景。

24.2.4 调停者模式的适用范围及优缺点

适用范围：

- 一组对象以定义良好但是复杂的方式进行通信的场合。
- 一个对象引用其他很多对象且直接与这些对象通信，导致难以复用该对象的场合。
- 定制一个分布在多个类中的行为，而又不想生成太多的子类的场合。

调停者模式的优点：

- 随着将所有对象的交互行为移到一个独立的对象中，用调停者的子类替换调停者或者改变它的功能可以很容易地改变对象之间内部的关联行为。
- 将对象的内部依赖关系移到一个单独的对象，这样会提高对象的可用性。
- 因为对象不需要直接引用其他的对象，所以对象可以更容易的进行单元测试。
- 类之间的低耦合可以使类在不影响其他类的基础上进行修改。

调停者模式的缺点：

- 调停者模式降低了同事对象的复杂性，代价是增加了调停者类的复杂性。当然，在很多情况下，设置一个调停者还是比设置一个同事类好。
- 调停者类中经常充斥着各个具体同事类的关系协调代码，这种代码常常是不能复用的。因此，具体同事类的复用是以调停者类的不可复用为代价的。

24.2.5 与其他模式的关系

调停者模式在定义上比较松散，在结构上和观察者模式、命令模式十分相像——都添加了中间件，只是调停者模式去掉了后两者在行为上的方向性；另外观察者模式中的观察者、命令模式中的命令都是被客户端所知的，是由客户来指定的，而大多调停者角色对于客户程序却是透明的，这样的差异是由于它们要达到的目的不同造成的。观察者模式适合用于消息的发现和监听，调停者模式更适合处理对象之间的复杂关系，所以两者可以合作使用。

24.3 调停者模式总结

调停者模式封装了一系列对象间相互作用的方式，使得这些对象间耦合度显著降低。从而当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用，保证了这些作用可以彼此独立地进行变化。调停者模式将多对多的相互作用转化为一对多的相互作用。将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

程序员在使用此模式时要特别关注其应用范围，使用不当反而会极大地增加系统的开销和编程的复杂度。

第25章 模板方法模式 (Template Method)

模板方法模式的设计意图是由抽象父类控制顶级逻辑，并把基本操作的实现推迟到子类中去实现，这是通过继承的手段来达到对对象的复用¹。

什么是模板方法模式？在回答这个问题前，咱们先来看看模板。提到模板，相信大家马上能够想到一些东西，如报表模板，报销单模板，简历模板等，使用它们的好处当然是显而易见的，它们可以给我们提供特定的结构和样式，我们只需关心要填充的数据内容即可。将模板的思想发散到编程上，就是模板方法模式了。

在实际编程过程中通常会遇到这样的情况：知道一个操作所需的关键步骤，并确定了这些步骤的执行顺序，但是某些步骤的具体实现是未知的，或者说某些步骤的实现与具体的环境相关。

模板方法模式把上面所讲的不知道的具体实现的步骤封装成抽象方法，提供一个按正确顺序调用它们的具体方法（这些具体方法统称为“模板方法”），从而构成一个抽象基类。子类通过继承这个抽象基类去实现各个步骤的抽象方法，而工作流程却由父类控制。

考虑一个简单的订单处理需求：一个客户可以在一个订货单中订购多个货物（也称为订货单项目），货物的售价是根据货物的进货价进行计算的（不同货物有不同的售价计算方法）。有些货物可以打折，有些是不可以打折的。每一个客户都有一个信用额度，每张订单的总金额不能超出该客户的信用额度。这样的订单处理过程将会有很多种具体的实现方式，但是操作步骤是一样的，这种情况就非常适合使用模板方法模式。

再比如说，一个人有一个日计划模板，其主要内容是：

- 早餐；
- 上午计划；
- 午餐；
- 下午计划；
- 晚餐；
- 晚上计划。

这个人会根据每一天的具体事项来填上具体的内容，每一天都会使用这样的模板，但是其中的内容是不同的，这也是一个典型的模板案例。

25.1 超市面包房的糕点制作流程规范化问题

超市中有一个独立的面包房，如图25-1所示。面包房可以制作各种糕点，做糕点的师傅有很多，糕点的种类也多种多样，因此制作过程十分复杂。

这里以面包的制作为例，一个典型的制作流程为：组建配方→材料称重→搅拌→基本发酵→分割→面团称重→滚圆→中间发酵→整形→装模→最后发酵→入炉烘烤→出炉→刷上光剂→

¹Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.GOF[95]

冷却→成品，如图25-2所示。图中，各个组件（按钮、工具栏、选项、菜单）之间都可能存在

千丝万缕的联系，比如在选择某种工具的时候某种操作不可以做，或者某种操作必须经过某种工具才能完成。这种操作是可以抽象为模板方法（Template Method）左翼去。

在网站设计中也经常遇到类似的情况，比如在未登录状态下不允许访问某些资源，或者在某些资源访问前必须先经过某种操作。这种操作是可以抽象为模板方法（Template Method）左翼去。

面包房



图25-1 面包房

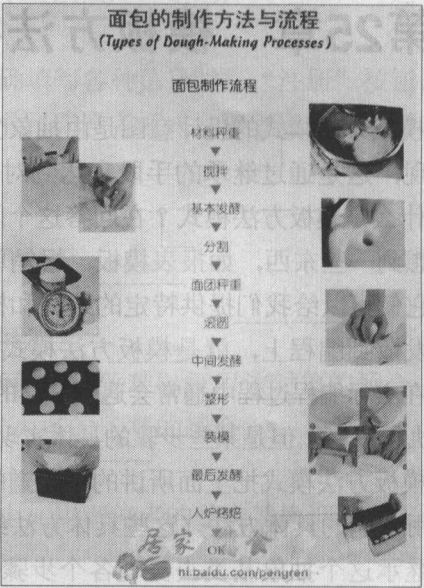


图25-2 面包的制作方法与部分流程

很明显，不同的面包在以上各个步骤中的情况都是有差异的，所以一定要在代码中显示其差异性。最简单的办法是将面包的制作过程硬编码进程序中。下面来看一下具体的实现代码。

抽象面包类：

代码片段1 Bread.java

```
1 package cn.steven.pattern.demo.templatemethod.quest;
2
3 /**
4  * 抽象面包类
5  */
6 abstract public class Bread {
7
8     /**
9      * 面包名称
10     */
11     private String name;
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     /**
22      * 抽象面包制作方法
23     */
```

```
24 abstract public void create();
25 }
```

水果面包类:

代码片段2 FruitBread.java

```
1 package cn.steven.pattern.demo.templateMethod.quest;
2
3 /**
4  * 水果面包
5  */
6 public class FruitBread extends Bread {
7
8     /**
9      * 构造方法
10     */
11     public FruitBread() {
12         super();
13         this.setName("水果面包");
14     }
15
16     /**
17      * 制作方法
18     */
19     @Override
20     public void create() {
21         /**
22          * 材料准备
23          */
24         System.out.println("准备水果");
25         System.out.println("准备面粉");
26
27         /**
28          * 处理材料
29          */
30         System.out.println("面团发酵");
31         System.out.println("加入水果");
32
33         /**
34          * 烘焙
35          */
36         System.out.println("低温烘焙");
37     }
38
39 }
```

牛奶面包类:

代码片段3 MilkBread.java

```
1 package cn.steven.pattern.demo.templateMethod.quest;
2
3 /**
```

```
4  * 牛奶面包
5  */
6  public class MilkBread extends Bread {
7
8      /**
9       * 构造方法
10      */
11     public MilkBread() {
12         super();
13         this.setName("牛奶面包");
14     }
15
16     /**
17     * 制作方法
18     */
19     @Override
20     public void create() {
21         /**
22         * 材料准备
23         */
24         System.out.println("准备牛奶");
25         System.out.println("准备面粉");
26
27         /**
28         * 处理材料
29         */
30         System.out.println("面团加入牛奶后发酵");
31
32         /**
33         * 烘焙
34         */
35         System.out.println("高温烘焙");
36     }
37 }
38
39 }
```

客户端代码:

代码片段4 Client.java

```
1  package cn.steven.pattern.demo.templateMethod.quest;
2
3  /**
4   * 客户端代码
5   */
6  public class Client {
7
8      public static void main(String[] args) {
9
10         /**
11         * 创建对象
```



```
12      */
13      Bread a = new FruitBread();
14      Bread b = new MilkBread();
15
16      /**
17       * 创造过程
18       */
19      a.create();
20      System.out.println(a.getName() + "制作好了!");
21      b.create();
22      System.out.println(b.getName() + "制作好了!");
23
24  }
25
26 }
```

代码的运行结果如图25-3所示。

由以上结果可见，通过这种编程方法的确可以实现面包制作的功能，但是考虑扩展性问题后，可以发现这种方法有以下缺陷：

- 面包的制作步骤中的细节可以不一样，但是步骤是一样的，不能共享。
- 特定步骤的细节不能共享。
- 面包制作的流程无法固定，实现类可以自由更改。

以上问题使用模板方法模式就可以解决。

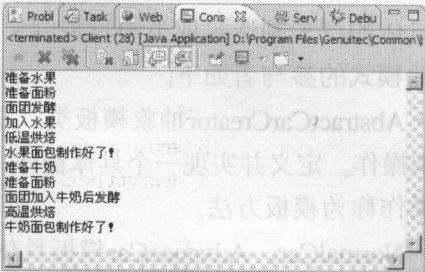


图25-3 运行结果

25.2 模板方法模式的结构

模板方法模式是通过继承的方式实现算法步骤的重用的，子类只需要重写特定的方法即可实现一个特定的流程。

25.2.1 模板方法模式

模板方法模式采用了继承的方式来实现步骤的重用，在某些面向对象的理论中，通常认为使用将接口实现和对象相聚合的方式来实现功能的重用会比继承好，但是这也要看实际情况，在可以使用模板方法模式解决的问题中，使用继承是很好的解决方案。

为了说明模板方法模式的实现方法，下面使用一个汽车制造的例子进行分析。比如有一个汽车流水线可以生产多种型号的汽车，如普通两厢车和豪华三厢车，这两种车的制作过程是一样的，都要先生产车架，再安轮胎，然后装挡位配件等。但是具体生产的细节是不一样的，比如普通车用普通轮胎，豪华车用加宽轮胎。

其设计类图如图25-4所示。

注意，图中AbstractCarCreator中定义有一个makeCar方法，此方法定义了制造汽车的顺序，无论以后生产什么汽车，都会按照此顺序来制作。

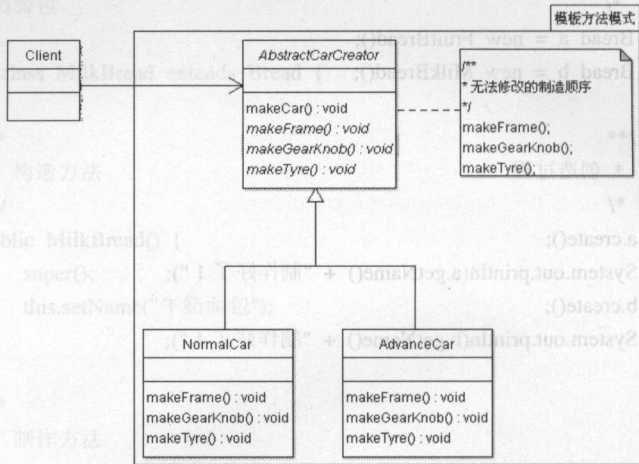


图25-4 模板方法模式类图

具体的汽车制造实现类中只需要实现创建特定零件部分的代码即可。
此模式的参与者如下。

- **AbstractCarCreator**抽象模板类：定义一个或多个抽象操作，由子类去实现。这些操作称为基本操作。定义并实现一个具体操作，这个具体操作通过调用基本操作控制顶级逻辑。这个具体操作称为模板方法。
- **NormalCar、AdvanceCar**模板具体类：实现抽象模板类所定义的抽象操作。
- **Client**客户端：调用模板类来描述具体的实现类对象如何进行操作。

此例的运行时序图如图25-5所示。

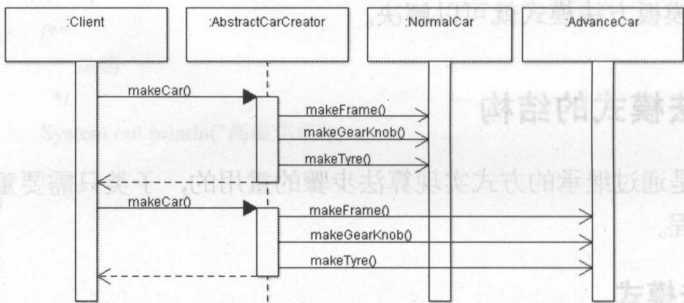


图25-5 运行时序图

下面展示实现代码，首先是抽象类：

代码片段5 AbstractCarCreator.java

```
1 package cn.steven.pattern.demo.templatemethod.pattern;
2
3 /**
4  * 抽象模板类，汽车制造
5  */
6 public abstract class AbstractCarCreator {
7
8     /**
9      * 制作方法被设计为模板方法，子类无法修改
```

```
10     */
11     public final void makeCar() {
12         /**
13          * 列出无法修改的制造顺序
14          */
15         makeFrame();
16         makeGearKnob();
17         makeTyre();
18     }
19
20     /**
21      * 抽象方法: 制作车架
22      */
23     abstract public void makeFrame();
24
25     /**
26      * 抽象方法: 制作挡位配件
27      */
28     abstract public void makeGearKnob();
29
30     /**
31      * 抽象方法: 制作轮胎
32      */
33     abstract public void makeTyre();
34 }
```

普通车实现类:

代码片段6 NormalCar.java

```
1 package cn.steven.pattern.demo.templateMethod.pattern;
2
3 /**
4  * 普通两厢车
5  */
6 public class NormalCar extends AbstractCarCreator {
7
8     /**
9      * 只需要实现具体零件的制造方法, 无法重写总体制造方法
10      */
11     @Override
12     public void makeFrame() {
13         System.out.println("安装了两厢车车架");
14     }
15
16     @Override
17     public void makeGearKnob() {
18         System.out.println("安装了手动挡");
19     }
20
21     @Override
22     public void makeTyre() {
```



```
23         System.out.println("安装了普通轮胎");
24     }
25
26 }
```

豪华车实现类:

代码片段7 AdvanceCar.java

```
1 package cn.steven.pattern.demo.templateMethod.pattern;
2
3 /**
4  * 豪华三厢车
5  */
6 public class AdvanceCar extends AbstractCarCreator {
7
8     /**
9      * 只需要实现具体零件的制造方法, 无法重写总体制造方法
10     */
11     @Override
12     public void makeFrame() {
13         System.out.println("安装了三厢车车架");
14     }
15
16     @Override
17     public void makeGearKnob() {
18         System.out.println("安装了自动挡");
19     }
20
21     @Override
22     public void makeTyre() {
23         System.out.println("安装了加宽轮胎");
24     }
25
26 }
```

客户端代码:

代码片段8 Client.java

```
1 package cn.steven.pattern.demo.templateMethod.pattern;
2
3 /**
4  * 客户端代码
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 创建普通汽车制造对象
12          */
13         AbstractCarCreator car1 = new NormalCar();
```

```
14
15  /**
16   * 制造
17   */
18  car1.makeCar(); bake();
19
20  System.out.println();
21
22  /**
23   * 创建豪华汽车制造对象
24   */
25  AbstractCarCreator car2 = new AdvanceCar();
26
27  /**
28   * 制造
29   */
30  car2.makeCar();
31
32  }
33
34 }
```

代码结果如图25-6所示。

结合代码和运行结果图可见，代码中使用了模板方法模式在抽象类中定义了一个不可被子类重写的方法来定义一个固定的执行过程，而具体的实现步骤由其子类实现，这样就很好地解决了固定的调用步骤和可变的调用实现之间的关系。

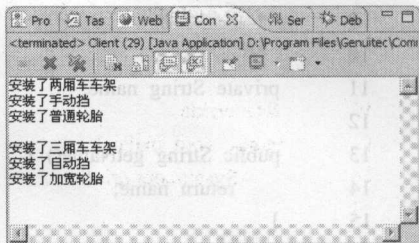


图25-6 运行结果

25.2.2 用模板方法模式设计不同糕点的制作流程

通过上一节内容的学习，相信读者已经初步了解了模板方法模式的设计过程，下面就可以解决糕点制作的问题了。

在下面的讲解中，除了满足原始的制作条件外还增加了一个最终包装的过程，因为各种不同的面包在制作出来后要根据客户的需求来进行包装，如果客户要马上吃掉面包，那么将面包直接放到盘子里交给顾客，如果顾客要打包带走就将面包放入包装袋中。

由需求可见，包装的需求其实和具体面包的制作方案是没有关系的，根据前面所学习的模式可以分析得出：使用装饰模式可以完成这一功能。下面就结合使用这两种模式来进行设计。

设计类图如图25-7所示。

下面展示实现代码，首先是最高层抽象类：

代码片段9 Bread.java

```
1 package cn.steven.pattern.demo.templatemethod;
2
3 /**
4  * 抽象面包类
5  */
```

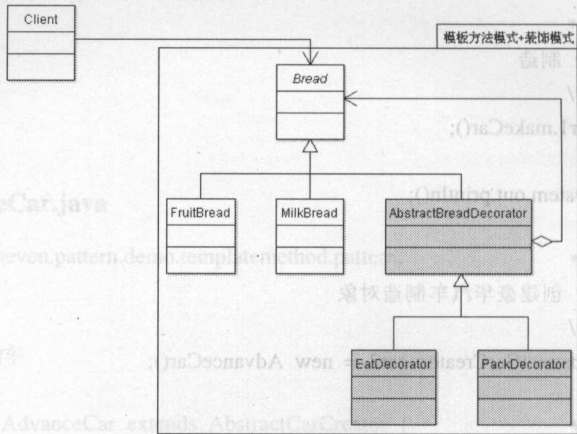


图25-7 模板方法模式和装饰模式的设计类图

```
6  abstract public class Bread {
7
8      /**
9       * 面包名称
10      */
11     private String name;
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     /**
22      * 模板方法：面包制作
23      * 没有定义成final方法以便被装饰模式使用
24      */
25     public void create() {
26         prepare();
27         handle();
28         bake();
29     }
30
31     /**
32      * 抽象方法：准备
33      */
34     abstract public void prepare();
35
36     /**
37      * 抽象方法：处理材料
38      */
39     abstract public void handle();
```



```

40  */
41  /** public class AbstractBreadDecorator extends Bread {
42      * 抽象方法: 烘焙
43      */
44      abstract public void bake();
45  }

```

水果面包:

代码片段10 FruitBread.java

```

1  package cn.steven.pattern.demo.templatemethod;
2
3  /**
4   * 水果面包
5   */
6  public class FruitBread extends Bread {
7
8      /** 委托这个模板方法
9       * 构造方法
10      */
11      @Override
12      public FruitBread() {
13          super();
14          this.setName("水果面包");
15      }
16
17      @Override
18      public void bake() {
19          /**
20           * 烘焙
21          */
22          System.out.println("低温烘焙");
23      }
24
25      @Override
26      public void handle() {
27          /**
28           * 处理材料
29          */
30          System.out.println("加入水果");
31      }
32
33      @Override
34      public void prepare() {
35          /**
36           * 材料准备
37          */
38          System.out.println("准备水果");
39      }
40
41  }

```

牛奶面包:

代码片段11 MilkBread.java

```
1 package cn.steven.pattern.demo.templateMethod;
2
3 /**
4  * 牛奶面包
5  */
6 public class MilkBread extends Bread {
7
8     /**
9      * 构造方法
10     */
11     public MilkBread() {
12         super();
13         this.setName("牛奶面包");
14     }
15
16     @Override
17     public void bake() {
18         /**
19          * 烘焙
20          */
21         System.out.println("高温烘焙");
22     }
23
24     @Override
25     public void handle() {
26         /**
27          * 处理材料
28          */
29         System.out.println("面团加入牛奶后发酵");
30     }
31
32     @Override
33     public void prepare() {
34         /**
35          * 材料准备
36          */
37         System.out.println("准备牛奶");
38     }
39
40 }
```

装饰抽象类:

代码片段12 AbstractBreadDecorator.java

```
1 package cn.steven.pattern.demo.templateMethod;
2
3 /**
4  * 抽象面包装饰类
```

```

5  */
6  abstract public class AbstractBreadDecorator extends Bread {
7
8      /**
9       * 聚合一个Bread对象
10      */
11     private Bread bread;
12
13     /**
14      * 构造方法
15      */
16     public AbstractBreadDecorator(Bread bread) {
17         super();
18         this.bread = bread;
19     }
20
21     /**
22      * 委托这个模板方法
23      */
24     @Override
25     public void create() {
26         bread.create();
27     }
28
29     @Override
30     public void bake() {
31         bread.bake();
32     }
33
34     @Override
35     public void handle() {
36         bread.handle();
37     }
38
39     @Override
40     public void prepare() {
41         bread.prepare();
42     }
43 }

```

直接吃的面包的包装类:

代码片段13 EatDecorator.java

```

1  package cn.steven.pattern.demo.templatemethod;
2
3  /**
4   * 当场可以吃的面包的装饰方法
5   */
6  public class EatDecorator extends AbstractBreadDecorator {
7
8      /**

```



```

9      * 构造方法
10     */
11     public EatDecorator(Bread bread) {
12         super(bread);
13     }
14
15     /**
16     * 重写模板方法
17     */
18     @Override
19     public void create() {
20         super.create();
21         System.out.println("将面包放于盘子中");
22     }
23
24 }

```

打包的包装类:

代码片段14 PackDecorator.java

```

1  package cn.steven.pattern.demo.templateMethod;
2
3  /**
4   * 打包面包的装饰方法
5   */
6  public class PackDecorator extends AbstractBreadDecorator {
7
8      /**
9      * 构造方法
10     */
11     public PackDecorator(Bread bread) {
12         super(bread);
13     }
14
15     /**
16     * 重写模板方法
17     */
18     @Override
19     public void create() {
20         super.create();
21         System.out.println("将面包包装起来");
22     }
23
24 }

```

客户端代码:

代码片段15 Client.java

```

1  package cn.steven.pattern.demo.templateMethod;
2
3  /**
4   * 客户端代码

```

```
5  */
6  public class Client {
7
8      public static void main(String[] args) {
9
10         /**
11          * 创建一个最后打包的牛奶面包
12          */
13         AbstractBreadDecorator breada = new PackDecorator(
14             new MilkBread());
15
16         /**
17          * 制作
18          */
19         breada.create();
20
21         System.out.println();
22
23         /**
24          * 创建一个最后直接吃的水果面包
25          */
26         AbstractBreadDecorator breadb = new EatDecorator(
27             new FruitBread());
28
29         /**
30          * 制作
31          */
32         breadb.create();
33     }
34 }
35
36 }
```

运行结果如图25-8所示。

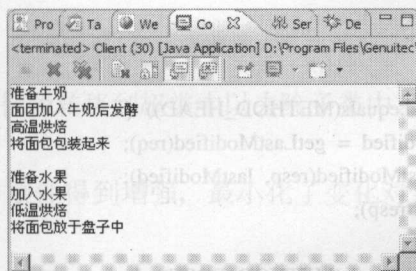


图25-8 运行结果

由运行结果可见，通过运用两种模式，现在已经可以自由地组合面包制作方式了。

25.2.3 模板方法模式在JDK中的实例

在JavaEE中HttpServlet是一个常用的类，此类中的service方法即为一个模板方法，客户访问Servlet时，会有七种访问方式分别对应于HttpServlet中的七个do方法，它们分别是：

- doGet;
- doHead;
- doPost;
- doPut;
- doDelete;
- doOptions;
- doTrace.

HttpServlet中定义的service方法就是用来判断需要具体调用哪一种do方法的，其代码如下所示：

代码片段16 HttpServlet.java节选

```

1  protected void service(HttpServletRequest req,
2      HttpServletResponse resp) throws ServletException,
3      IOException {
4
5      String method = req.getMethod();
6
7      if (method.equals(METHOD_GET)) {
8          long lastModified = getLastModified(req);
9          if (lastModified == -1) {
10             doGet(req, resp);
11         } else {
12             long ifModifiedSince = req
13                 .getDateHeader(HEADER_IFMODSINCE);
14             if (ifModifiedSince < (lastModified / 1000 * 1000)) {
15                 maybeSetLastModified(resp, lastModified);
16                 doGet(req, resp);
17             } else {
18                 resp
19                     .setStatus(
20                         HttpServletResponse.SC_NOT_MODIFIED);
21             }
22         }
23
24     } else if (method.equals(METHOD_HEAD)) {
25         long lastModified = getLastModified(req);
26         maybeSetLastModified(resp, lastModified);
27         doHead(req, resp);
28
29     } else if (method.equals(METHOD_POST)) {
30         doPost(req, resp);
31
32     } else if (method.equals(METHOD_PUT)) {
33         doPut(req, resp);
34
35     } else if (method.equals(METHOD_DELETE)) {
36         doDelete(req, resp);
37

```



```

38     } else if (method.equals(METHOD_OPTIONS)) {
39         doOptions(req, resp);
40
41     } else if (method.equals(METHOD_TRACE)) {
42         doTrace(req, resp);
43
44     } else {
45
46         String errMsg = IStrings
47             .getString("http.method_not_implemented");
48         Object[] errArgs = new Object[1];
49         errArgs[0] = method;
50         errMsg = MessageFormat.format(errMsg, errArgs);
51
52         resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED,
53             errMsg);
54     }
55 }

```

由代码可见service方法已经定义了一个算法的顺序, 这样, 一个具体的Servlet就无需再定义自己的执行顺序了。但是需要注意的是, HttpServlet中定义的service方法是可以被重写的, 以方便扩展功能。

在JavaSE中也有不少使用模板方法模式的例子, 如javax.swing包中的AbstractTableModel类和其他表示模型的类。

25.2.4 模板方法模式的使用范围及优点

模板方法的设计分类:

- 抽象模板角色里提供完整的方法, 它完成了所有派生类都要用到的一些基本功能。
- 抽象模板角色里只提供空方法, 把功能全部留给派生类去实现。
- 抽象模板角色里只包含某些操作的默认实现, 派生类里可以重新定义这些方法的实现。
- 抽象模板角色里的模板方法, 它是一个调用抽象方法、钩子方法以及具体方法的各种组合。

优点:

- 模板方法模式把不变的行为转移到超类中以去除子类中重复的代码, 从而提供了很好的代码复用平台。

- 使用模板方法使系统扩展性得到增强, 最小化了变化对系统的影响。

缺点:

- 会增加很多具体方法的数量。
- 如果选用的实现方式不当, 复用情况会很差。

应用范围:

- 子类具有统一的操作步骤或操作过程;
- 子类具有不同的操作细节;
- 存在多个具有同样操作步骤的应用场景, 但某些具体的操作细节却各不相同。

25.2.5 与其他模式的关系

与策略模式相比，模板方法模式的中心放在了方法调用的顺序上，策略模式的中心集中在方法的封装上。模板方法模式的子类不需要改变一个算法的结构就可以改变一个算法的特定步骤。

模板方法模式可以和装饰模式结合使用以扩展不属于原对象功能体系的功能。

模板方法模式经常会调用工厂方法模式来生成需要的对象。

25.3 模板方法模式总结

“不要给我们打电话，我们会给你打电话”这是著名的好莱坞招聘原则。在好莱坞，把简历递交给演艺公司后你就只能回家等待。由演艺公司对整个项目进行完全控制，演员只能被动地接受公司的指挥。模板方法模式充分体现了“好莱坞”原则，它由父类完全控制子类的逻辑，这就是控制反转。子类可以实现父类的可变部分，但要继承父类的逻辑，且不能改变业务逻辑。这就是模板方法模式的实现思想。

在实际应用中，应充分发挥此模式的复用能力，对于多个子类都可以使用的方法，可以在抽象类中创建一个默认的实现，这样可以减少其编程复杂度。

```
11 } else {
12     long lastModifiedSince = req
13         .getDateHeader("If-Modified-Since");
14     if (lastModifiedSince > 0) {
15         maybeSetLastModified(resp, lastModified);
16     }
17     // 这里应该调用 doPost 方法
18     doPost(req, resp);
19 }
20
21 }
22
23
24
25 long lastModified = getLastModified(req);
26 maybeSetLastModified(resp, lastModified);
27 doPost(req, resp);
28
29 } else if (method.equals(METHOD_POST)) {
30     doPost(req, resp);
31 } else if (method.equals(METHOD_PUT)) {
32     doPut(req, resp);
33 } else if (method.equals(METHOD_DELETE)) {
34     doDelete(req, resp);
35 }
```

第26章 解释器模式 (Interpreter)

解释器模式给定一种语言，定义它的文法标识，并定义一个解释器，然后这个解释器将使用该文法标识来解释语言中的句子¹。

在软件构建过程中，如果某一特定领域的问题比较复杂，类似的结构不断重复出现，而使用普通的编程方式来实现将面临非常频繁的变化。在这种情况下，将特定领域的问题表达为某种语法规则下的句子，然后构建一个解释器来解释这样的句子，可以达到解决问题的目的。

在日常生活中也会出现这样的例子，最普遍的就是语言翻译，比如将一种语言翻译为另一种语言时，同样的词汇将会反复地出现，如果用普通的编程方式，编码中将会出现大量的判读语句，而且可扩展性很差，这种情况下就适合使用解释器模式。

26.1 新店开在了外国人聚居区

由于使用了各种模式来解决超市的营业问题，超市的连锁店越开越多。目前，超市经理想在一个外国人的集中居住区开设连锁店，但是问题是超市里的文字指示牌需要设置多国语言的译文。

一种普通的做法是将固定的语句匹配列出，每次出现需要翻译的句子时去其中进行查找。下面来看一下代码的设计过程，比如将中文翻译为英语、日语和韩语：

代码片段1 InterpretationTool.java

```
1 package cn.steven.pattern.demo.interpreter.quest;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * 翻译工具
8  */
9 public class InterpretationTool {
10
11     /**
12      * 存放翻译的数据
13      */
14     private Map<String, Map<String, String>> map;
15
16     public Map<String, Map<String, String>> getMap() {
17         return map;
18     }
19
20     public void setMap(Map<String, Map<String, String>> map) {
```

¹Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.GOF[95]


```
21         this.map = map;
22     }
23
24     /**
25      * 构造方法
26      */
27     public InterpretationTool() {
28         super();
29     }
30     /**
31      * 构建翻译数据
32      */
33     map = new HashMap<String, Map<String, String>>();
34
35     /**
36      * 放入英、日、韩三国的翻译数据
37      */
38     Map<String, String> m = new HashMap<String, String>();
39
40     /**
41      * 英语数据
42      */
43     m.put("en", "good morning");
44
45     /**
46      * 日语数据
47      */
48     m.put("ja", " == 0 1 == ");
49
50     /**
51      * 韩语数据
52      */
53     m.put("ko", " == 0 2 == ");
54
55     /**
56      * 翻译数据存入表中
57      */
58     map.put("早上好", m);
59 }
60
61 /**
62  * 翻译方法
63  *
64  * @param key
65  *         要翻译的中文
66  * @param lang
67  *         翻译成的语言
68  * @return
69  */
70 public String translate(String key, String lang) {
71     try {
72         Map<String, String> m = map.get(key);
```

```
72         String c = m.get(lang);
73         if (c != null) {
74             return c;
75         }
76     } catch (Exception e) {
77     }
78     System.out.println("找不到" + lang + "语言的 " + key);
79     return null;
80 }
81
82 }
```

客户端类:

代码片段2 Client.java

```
1 package cn.steven.pattern.demo.interpreter.quest;
2
3 /**
4  * 翻译客户端
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10         /**
11          * 需要翻译的语句
12          */
13         String content = "早上好";
14
15         /**
16          * 得到翻译
17          */
18         InterpretationTool tool = new InterpretationTool();
19         System.out.println("日语:"+tool.translate(content, "ja"));
20         System.out.println("英语:"+tool.translate(content, "en"));
21         System.out.println("韩语:"+tool.translate(content, "ko"));
22         System.out.println("德语:"+tool.translate(content, "de"));
23     }
24
25 }
```

代码运行后如图26-1所示。

由代码和图26-1可见，这种编程方式可以解决翻译特定语句的问题，但是当翻译语句的规模扩大之后就会发生非常严重的问题：

- 只能翻译特定的语句，无法根据词汇组装成句，不智能。
- 增加一种语言处理起来极为复杂。
- 语法不具备复用性。

使用解释器模式可以解决这些问题。

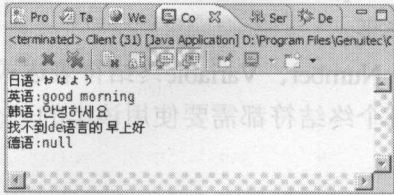


图26-1 运行结果

26.2 解释器模式的结构

解释器模式的结构非常像组合模式，但是它们处理问题的出发点是不同的。

26.2.1 解释器模式

为了说明解释器模式的实现方法，下面的例子将编写一个逆波兰表示法¹的解释器。

逆波兰记法

逆波兰记法中，操作符置于操作数的后面。例如表达“三加四”时，写作“3 4 +”，而不是“3 + 4”。如果有多个操作符，操作符置于第二个操作数的后面，所以常规中缀记法的“3 - 4 + 5”，在逆波兰记法中写做“3 4 - 5 +”：先3减去4，再加上5。使用逆波兰记法的一个好处是不需要使用括号。例如，中缀记法中“3 - 4 * 5”与“(3 - 4) * 5”不相同，但后缀记法中前者写做“3 4 5 * -”，无歧义地表示法为“3 (4 5 *) ?”；后者写做“3 4 - 5 *”。

逆波兰表达式的解释器一般是基于堆栈的。解释过程一般是：操作数入栈；遇到操作符时，操作数出栈，求值，将结果入栈；当执行一遍后，栈顶就是表达式的值。因此逆波兰表达式的求值使用堆栈结构很容易实现，并能很快求值。

注意：逆波兰记法并不是简单的波兰表达式的反转。因为对于不满足交换律的操作符，它的操作数写法仍然是常规顺序，如，波兰记法“/ 6 3”的逆波兰记法是“6 3 /”而不是“3 6 /”；数字的数位写法也是常规顺序。

逆波兰标记法中元素的定义如下：

- 表达式 `expression ::= plus | minus | variable`;
- 加法 `plus ::= expression expression '+'`;
- 减法 `minus ::= expression expression '-'`;
- 变量 `variable ::= 'a' | 'b' | 'c' | ... | 'z'`。

如果有下述表达式：

- `a b +`
- `a b c + -`
- `a b + c a - -`

计算机在知道逆波兰算法之后将如何解释其执行过程呢？使用解释器模式的设计类图如图26-2所示。

其中的参与者分别为：

- **Expression** 表达式接口：声明一个抽象的解释操作，这个接口为抽象语法树中所有的节点所共享。

- **Number、Variable** 终结符表达式：实现与文法中的终结符相关联的解释操作，一个句子中的每个终结符都需要使用该类的一个实例。

¹http://en.wikipedia.org/wiki/Reverse_Polish_notation.

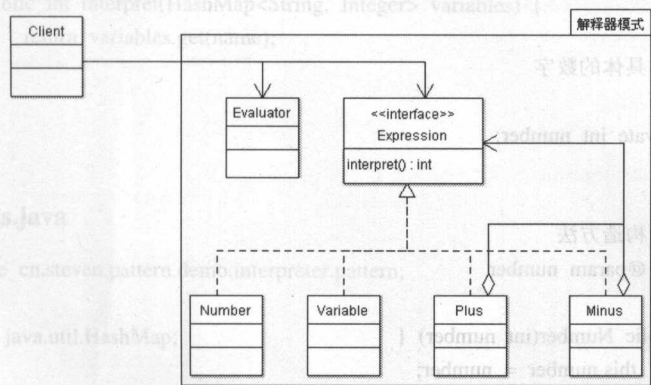


图26-2 解释器模式类图

- **Plus、Minus非终结符表达式**：对于文法中的每一条规则都需要一个非终结表达式类，来为文法中的非终结符实现解释操作，解释操作一般要递归地调用表示之前那些对象的解释操作。
- **Evaluator运算器**：构建表示该文法定义的语言中一个特定的句子的抽象语法树。
- **Client客户端**：输入需要解释的表达式，调用运算器进行解释动作。

下面看一下具体的实现代码，首先是表达式接口：

代码片段3 Expression.java

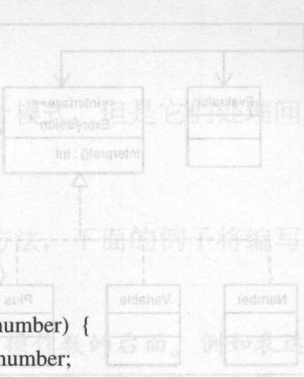
```
1 package cn.steven.pattern.demo.interpreter.pattern;
2
3 import java.util.HashMap;
4
5 /**
6  * 解释器模式表达式
7  */
8 public interface Expression {
9
10     /**
11      * 解释方法
12      * @param variables 使用的变量
13      * @return
14      */
15     public int interpret(HashMap<String, Integer> variables);
16 }
```

数字表达式：

代码片段4 Number.java

```
1 package cn.steven.pattern.demo.interpreter.pattern;
2
3 import java.util.HashMap;
4
5 /**
6  * 数字类型
7  */
8 class Number implements Expression {
```

```
9
10 /**
11  * 具体的数字
12  */
13 private int number;
14
15 /**
16  * 构造方法
17  * @param number
18  */
19 public Number(int number) {
20     this.number = number;
21 }
22
23 /**
24  * 重写的解释方法
25  */
26 public int interpret(HashMap<String, Integer> variables) {
27     return number;
28 }
29 }
```



变量表达式:

代码片段5 Variable.java

```
1 package cn.steven.pattern.demo.interpreter.pattern;
2
3 import java.util.HashMap;
4
5 /**
6  * 变量解释器
7  */
8 class Variable implements Expression {
9
10     /**
11      * 变量名称
12      */
13     private String name;
14
15     /**
16      * 构造方法
17      *
18      * @param name
19      */
20     public Variable(String name) {
21         this.name = name;
22     }
23
24     /**
25      * 重写的解释方法
26      */
```

```
1 import java.util.HashMap;
2
3 /**
4  * 解释器接口
5  */
6 public interface Expression {
7
8     /**
9      * 解释方法
10      *
11      * @param variables 使用的变量
12      * @return
13      */
14     int interpret(HashMap<String, Integer> variables);
15 }
16
17 /**
18  * 数字表达式
19  */
20 class Number implements Expression {
21
22     /**
23      * 构造方法
24      *
25      * @param number
26      */
27     public Number(int number) {
28         this.number = number;
29     }
30
31     /**
32      * 重写的解释方法
33      */
34     int interpret(HashMap<String, Integer> variables) {
35         return number;
36     }
37 }
```

```

27     public int interpret(HashMap<String, Integer> variables){
28         return variables.get(name);
29     }
30 }

```

加法表达式:

代码片段6 Plus.java

```

1  package cn.steven.pattern.demo.interpreter.pattern;
2
3  import java.util.HashMap;
4
5  /**
6   * 加法解释器
7   */
8  class Plus implements Expression {
9
10     /**
11      * 左值
12      */
13     Expression leftOperand;
14
15     /**
16      * 右值
17      */
18     Expression rightOperand;
19
20     /**
21      * 构造方法
22      * @param left
23      * @param right
24      */
25     public Plus(Expression left, Expression right) {
26         leftOperand = left;
27         rightOperand = right;
28     }
29
30     /**
31      * 重写的解释方法
32      */
33     public int interpret(HashMap<String, Integer> variables) {
34         return leftOperand.interpret(variables)
35             + rightOperand.interpret(variables);
36     }
37 }

```

减法表达式:

代码片段7 Minus.java

```

1  package cn.steven.pattern.demo.interpreter.pattern;
2

```



```
3 import java.util.HashMap;
4
5 /**
6  * 减法解释器
7  */
8 class Minus implements Expression {
9
10     /**
11      * 左值
12      */
13     Expression leftOperand;
14
15     /**
16      * 右值
17      */
18     Expression rightOperand;
19
20     /**
21      * 构造方法
22      *
23      * @param left
24      * @param right
25      */
26     public Minus(Expression left, Expression right) {
27         leftOperand = left;
28         rightOperand = right;
29     }
30
31     /**
32      * 重写的解释方法
33      */
34     public int interpret(HashMap<String, Integer> variables) {
35         return leftOperand.interpret(variables)
36             - rightOperand.interpret(variables);
37     }
38 }
```

计算器:

代码片段8 Evaluator.java

```
1 package cn.steven.pattern.demo.interpreter.pattern;
2
3 import java.util.HashMap;
4 import java.util.Stack;
5
6 /**
7  * 计算器
8  */
9 public class Evaluator {
10
11     /**
```

```

12     * 表达式树
13     */
14     private Expression syntaxTree;
15
16     /**
17     * 构造方法
18     * @param expression
19     */
20     public Evaluator(String expression) {
21
22         /**
23          * 使用堆栈处理表达式
24          */
25         Stack<Expression> expressionStack = new Stack<Expression>();
26
27         /**
28          * 分割表达式
29          */
30         for (String token : expression.split(" ")) {
31             if (token.equals("+")) {
32                 Expression subExpression = new Plus(expressionStack
33                     .pop(), expressionStack.pop());
34                 expressionStack.push(subExpression);
35             } else if (token.equals("-")) {
36                 Expression subExpression = new Minus(expressionStack
37                     .pop(), expressionStack.pop());
38                 expressionStack.push(subExpression);
39             } else
40                 expressionStack.push(new Variable(token));
41         }
42         syntaxTree = expressionStack.pop();
43     }
44
45     /**
46     * 计算方法
47     * @param context
48     * @return
49     */
50     public int evaluate(HashMap<String, Integer> context) {
51         return syntaxTree.interpret(context);
52     }
53 }

```

客户端代码:

代码片段9 Client.java

```

1 package cn.steven.pattern.demo.interpreter.pattern;
2
3 import java.util.HashMap;
4
5 /**

```

```
6  * 解释器模式客户端
7  */
8  public class Client {
9
10     public static void main(String[] args) {
11
12         /**
13          * 给出表达式
14          */
15         String expression = "x y z - +";
16
17         /**
18          * 创建计算器
19          */
20         Evaluator sentence = new Evaluator(expression);
21         HashMap<String, Integer> variables
22             = new HashMap<String, Integer>();
23
24         /**
25          * 变量赋值
26          */
27         variables.put("x", 5);
28         variables.put("y", 8);
29         variables.put("z", 11);
30         System.out.println("x = 5\ny = 8\nz = 11\n");
31
32         /**
33          * 计算结果
34          */
35         int result = sentence.evaluate(variables);
36         System.out.println(expression + " = " + result);
37     }
38 }
39
40 }
```

运行结果如图26-3所示。

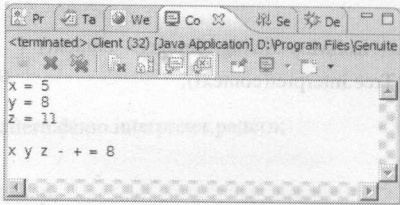


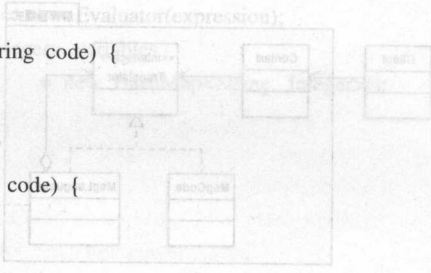
图26-3 解释器模式运行结果

由代码和结果可见，使用解释器模式很好地解决了逆波兰表示法的运算问题，且如果要增加新的运算符时也可以很方便地编写类加入进去，并不会影响原有的代码功能。

26.2.2 用解释器模式设计超市内的多语言指示牌

通过上一节的学习，对于解释器模式的设计相信读者已经可以初步掌握了，下面再来回顾


```
2
3 /**
4  * 消息编码
5  */
6 public class MsgCode implements Translator {
7
8     /**
9      * 存放的编号
10     */
11     private String code;
12
13     public String getCode() {
14         return code;
15     }
16
17     public void setCode(String code) {
18         this.code = code;
19     }
20
21     /**
22      * 翻译方法
23      */
24     @Override
25     public String translate(Context ctx) {
26         return code;
27     }
28 }
29
30
31
32
33
34 }
```



语言代码翻译类:

代码片段12 MsgLanguage.java

```
1 package cn.steven.pattern.demo.interpreter;
2
3 import java.util.Locale;
4 import java.util.ResourceBundle;
5
6 /**
7  * 消息语言
8  */
9 public class MsgLanguage implements Translator {
10
11     /**
12      * 语言资源文件路径
13     */
14     private String bName = "cn.steven.pattern.demo.interpreter.msg";
15 }
```

```

15
16 /**
17  * 存放需要操作的翻译内容
18  */
19 private Translator trans;
20
21 /**
22  * 存放需要翻译的语言
23  */
24 private String lang;
25
26 public String getLang() {
27     return lang;
28 }
29
30 public void setLang(String lang) {
31     this.lang = lang;
32 }
33
34 public Translator getTrans() {
35     return trans;
36 }
37
38 public void setTrans(Translator trans) {
39     this.trans = trans;
40 }
41
42 /**
43  * 构造方法
44  *
45  * @param trans
46  * @param lang
47  */
48 public MsgLanguage(Translator trans, String lang) {
49     super();
50     this.trans = trans;
51     this.lang = lang;
52 }
53
54 /**
55  * 翻译方法
56  */
57 @Override
58 public String translate(Context ctx) {
59
60     /**
61      * 绑定特定的资源文件
62      */
63     ResourceBundle res = ResourceBundle.getBundle(bName,
64         new Locale(lang));
65

```



```
66     /**
67     * 返回翻译结果
68     */
69     return res.getString(trans.translate(ctx));
70 }
71 }
```

语言代码翻译类中将会使用到ResourceBundle类，此类依赖于外界的语言properties文件来翻译特定的消息，其中文配置文件的内容如下：

代码片段13 msg_zh.properties

```
1 #中文配置文件
2 001=\u4f60\u597d
3 002=\u65e9\u4e0a\u597d
```

英文配置文件：

代码片段14 msg_en.properties

```
1 #英文配置文件
2 001=hello
3 002=good morning
```

日文配置文件：

代码片段15 msg_ja.properties

```
1 #日文配置文件
2 001=\u3053\u3093\u306b\u3061\u306f
3 002=\u304a\u306f\u3088\u3046
```

上下文环境类：

代码片段16 Context.java

```
1 package cn.steven.pattern.demo.interpreter;
2
3 /**
4  * 上下文环境
5  */
6 public class Context {
7
8     /**
9     * 解析
10    */
11    public String translator(String expression) {
12
13        /**
14        * 分割表达式
15        */
16        String[] split = expression.split(" ");
17
18        /**
19        * 构造解析对象
```

```
20      */
21      Translator msgCode = new MsgCode(split[0]);
22      Translator msgLanguage = new MsgLanguage(msgCode, split[1]);
23
24      /**
25       * 解析
26       */
27      return msgLanguage.translate(this);
28  }
29 }
```

客户端代码:

代码片段17 Client.java

```
1 package cn.steven.pattern.demo.interpreter;
2
3 /**
4  * 多国语言客户端
5  */
6 public class Client {
7
8     public static void main(String[] args) {
9
10        /**
11         * 创建上下文
12         */
13        Context ctx = new Context();
14
15        /**
16         * 执行测试
17         */
18        System.out.println("001 zh =>" + ctx.translator("001 zh"));
19        System.out.println("001 en =>" + ctx.translator("001 en"));
20        System.out.println("001 ja =>" + ctx.translator("001 ja"));
21
22        System.out.println("002 zh =>" + ctx.translator("002 zh"));
23        System.out.println("002 en =>" + ctx.translator("002 en"));
24        System.out.println("002 ja =>" + ctx.translator("002 ja"));
25    }
26
27 }
```

运行结果如图26-5所示。

由上述代码和运行结果可见，通过解释器模式的设计，多语言指示牌的问题已经可以很好地解决了，如果以后有新的语言或语句出现的话，只需要修改各个properties资源文件即可实现功能的扩充，从而极大地方便了客户的使用。

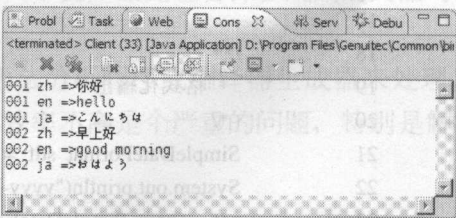


图26-5 运行结果

26.2.3 解释器模式在JDK中的实例

JDK中的SimpleDateFormat类，可以根据你给定的规则（如mm/dd/yyyy表示yyyy年MM月dd日），把一个字符串翻译成一个日期。查看JDK API可得到如下内容：

SimpleDateFormat是一个以与语言环境有关的方式来格式化和解析日期的具体类。它允许进行格式化（日期->文本）、解析（文本->日期）和规范化。

SimpleDateFormat使得人们可以选择任何用户定义的日期-时间格式的模式。但是，仍然建议通过DateFormat中的getTimeInstance、getDateInstance或getDateTimeInstance来创建日期-时间格式器。每一个这样的类方法都能够返回一个以默认格式模式初始化的日期/时间格式器。可以根据需要使用ApplyPattern方法来修改格式模式。有关使用这些方法的更多信息，请参阅DateFormat。

日期和时间模式

日期和时间格式由日期和时间模式字符串指定。在日期和时间模式字符串中，未加引号的字母‘A’~‘Z’和‘a’~‘z’被解释为模式字母，用来表示日期或时间字符串元素。文本可以使用单引号（'）括起来，以免进行解释。“”表示一对单引号。所有其他字符均不解释；只是在格式化时将它们简单复制到输出字符串，或者在解析时与输入字符串进行匹配。

SimpleDateFormat中定义的规则很多，所以在使用的时候通常需要查询手册，下面展示一下通常使用的方式：

代码片段18 SimpleDateFormat的使用方法Client.java

```
1 package cn.steven.pattern.demo.interpreter.jdk;
2
3 import java.text.SimpleDateFormat;
4 import java.util.Date;
5
6 /**
7  * SimpleDateFormat 举例
8  */
9 public class Client {
10
11     public static void main(String[] args) {
12
13         /**
14          * 创建时间对象
15          */
16         Date d = new Date();
17
18         /**
19          * 格式化输出
20          */
21         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
22         System.out.println("yyyy-MM-dd => " + sdf.format(d));
23         sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss.S Z");
24         System.out.println("yyyy/MM/dd HH:mm:ss.S Z => "
25             + sdf.format(d));
26         sdf = new SimpleDateFormat("yyyy年M月d日 H时m分s秒 S毫秒");
```



```

27 System.out.println("yyyy年M月d日 H时m分s秒 S毫秒 => "
28 + sdf.format(d));
29 sdf = new SimpleDateFormat("您所在的时区是: Z");
30 System.out.println("您所在的时区是: Z => " + sdf.format(d));
31
32 }
33 }

```

运行结果如图26-6所示。

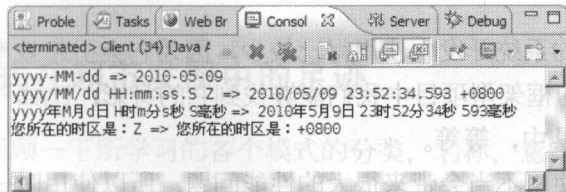


图26-6 SimpleDateFormat运行结果

由上面的示例可见，SimpleDateFormat中使用的解释器模式的确可以有效地满足表达式运算的要求，由于此类中的代码非常复杂，感兴趣的读者可以自行查看其源代码进行学习。

26.2.4 解释器模式的使用范围及优点

解释器模式的应用范围：

- 当有一种语言需要解释执行，并且可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。
- 对于复杂的文法，文法的类层次会变得庞大而无法管理，此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式，这样可以节省空间而且还可节省时间。
- 效率不是一个关键问题，最高效的解释器通常不是通过直接解释语法分析树实现的，而是首先将它们转换成另一种形式。例如：正则表达式通常被转换成状态机，但即使在这种情况下，转换器仍可用解释器模式实现，该模式仍是有用的。

解释器模式的优点：

- 能较容易地改变和扩展语法，该模式使用类来表示语法规则，程序员可使用继承来改变或扩展该文法。
- 比较容易实现文法，因为定义抽象语法树中各个节点的类的实现大体类似。

解释器模式的缺点：

- 解释器模式要为文法中的每一条规则至少定义一个类，因此包含许多规则的文法可能难以管理和维护。
- 当文法很复杂时，最好使用其他的技术，如语法分析程序或编译器生成器来处理。
- 解释器模式使用了大量的循环和递归，因此其效率不高是个严重的问题，特别是解析复杂、冗长的语法时，效率很低。

26.2.5 与其他模式的关系

与命令模式一样，解释器模式也会产生一个可执行的对象。差别在于解释器模式会创建一

解释器模式也类似于组合模式。组合模式通常会为单个对象和群组对象定义一个公共接口。不过，组合模式并不要求支持以不同方式组织的结构，尽管该模式可以支持这些结构。

由于解释器模式的使用不当会带来很大的性能问题，所以使用时一定要加以斟酌，分析较成熟的语法时可以先查找相关的开源框架工具包进行参考。

第27章 设计模式总结

经过了漫长的26章内容的学习历程，至此GOF的23种设计模式已经学习完了，但是这些模式其实只是软件设计模式中最基础的部分，我们学习的道路任重而道远。下面我们来对学习的内容进行总结以及明确一下未来模式学习的方向。

27.1 回顾设计模式在超市发展中的足迹

下面让我们一起来回顾一下所学习的各个模式的分类、名称、意图以及它解决的是超市中的什么问题，如表27-1所示。

表27-1 设计模式汇总

	模式名称	模式意图	解决超市的问题
创建型模式	工厂方法模式	工厂方法模式定义了一个用户创建的对象的接口，让子类决定实例化哪一个类。工厂方法模式使一个类的实例化延迟到其子类	商品上架问题
	抽象工厂模式	抽象工厂模式提供一个创建一系列相关或相互依赖的对象的接口，而无需指定它们的具体类	商品上架问题
	建造者模式	建造者模式将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示	捆绑销售问题
	原型模式	原型模式使用原型实例创建对象的种类，并通过拷贝这些原型创建新的对象	新开连锁店问题
	单例模式	单例模式确保某一个类只有一个实例，而且能自行实例化并向整个系统提供这个实例。这个类称为单例类	连锁店总店地位的唯一性问题
结构型模式	适配器模式	适配器模式在编程过程中是一个经常用到的模式，它的作用是将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作	电源插头问题
	桥接模式	桥接模式将抽象部分与实现部分分离，使它们都可以独立变化	游戏机销售的问题
	代理模式	代理模式为其他对象提供一种代理以控制对这个对象的访问	手机展示过程中的问题
	外观模式	为了给子系统中的一组接口提供一个一致的界面，该模式定义了一个高层接口，这个接口使得这一子系统更加容易使用	顾客的一站式服务问题

(续表)

	模式名称	模式意图	解决超市的问题
	装饰模式	装饰模式可以动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活	销售人员能力培养问题
	组合模式	组合模式将对象组合成树形结构以表示“部分-整体”的层次结构，它使得用户对单个对象和组合对象的使用具有一致性	超市组织结构问题
	享元模式	享元模式的意图是用共享技术有效地支持大量的细粒度对象	宣传海报的设计问题
行为型模式	命令模式	命令模式的意图是将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化、对请求进行排队或记录请求的日志，以及支持可以撤销的操作	客服电话的问题
	观察者模式	观察者模式属于行为型模式，其意图是定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新	连锁店宣传资料的发放问题
	责任链模式	该模式使多个对象都有机会处理请求（Request），从而避免请求的发送者和接收者之间的耦合关系	建立明晰的进货审批
	迭代器模式	迭代器模式提供一种方法来访问一个容器（container）对象中的各个元素，而又可以不暴露该对象的内部细节	统一处理各类商品
	访问者模式	访问者模式表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作	设计灵活高效的收银程序
	状态模式	状态模式允许一个对象在其内部状态改变时改变它的行为，令对象看起来就像是修改了它的类	设计超市在不同时段的打折状态
	备忘录模式	备忘录模式在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态	设计可靠的数据保存系统
	策略模式	策略模式属于行为模式，其意图是定义一系列算法，把它们一个个封装起来，并且使它们可以互相替换，从而使得算法可以独立于使用它的客户而变化	构建灵活的打折方式
	调停者模式	调停者模式定义了一个可以封装一组对象之间的相互影响的行为的对象，这种方式可以以松耦合的方式来联系一组对象，避免对象之间互相显式直接引用，这样当改变了某些对象之间的关系时，可以不影响其他的对象	协调进货/销售/盘点之间的关系

(续表)

模式名称	模式意图	解决超市的问题
模板方法模式	模板方法模式的意图是由抽象父类控制顶级逻辑，并把基本操作的实现推迟到子类去实现，其中通过继承的手段来达到对象的复用	设计不同糕点的制作流程
解释器模式	解释器模式将给定一种语言，然后定义它的文法标识，并定义一个解释器，这个解释器使用该标识来解释语言中的句子	设计超市内的多语言指示牌

通过表27-1可见，学习设计模式的历程也就是实现一个超市软件的过程。在学习的时候，读者不只要理解模式的意图和构建方法，还要学会怎样使用模式解决现实中的软件问题，这样才能真正学以致用。

27.2 各模式之间的关系及演变图

各个设计模式之间都是相互联系的，这些联系有相互合作的，有相互竞争的，还有结合使用的。在具体的使用过程中，人们通常都会将一系列模式按实际所需结合使用。

GoF的23种设计模式间的关系如图27-1所示。

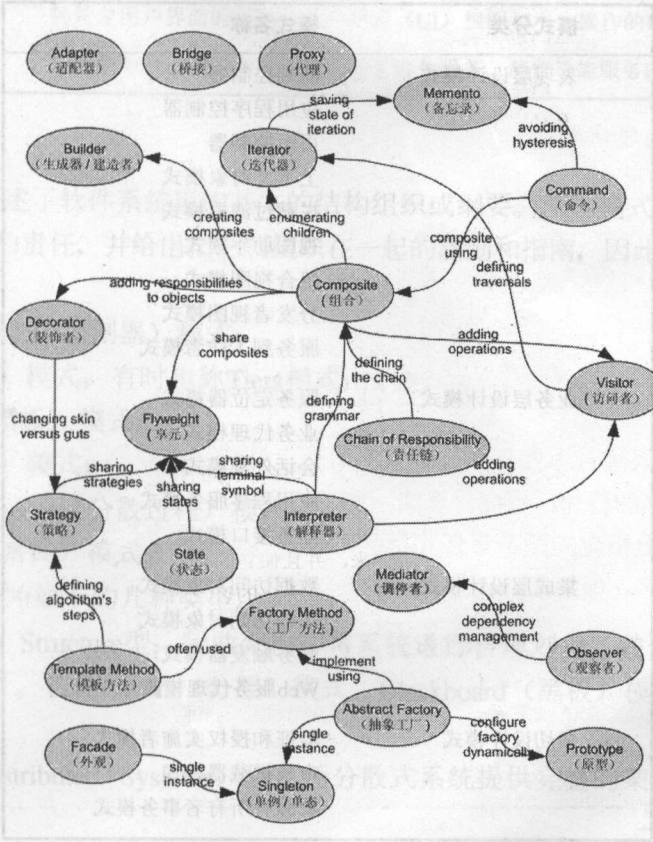


图27-1 模式关系图

由图27-1可见，设计模式中有几个核心的模式：

- 组合模式。
- 迭代器模式。
- 抽象工厂模式。
- 访问者模式。

在学习模式时可以先学习这几个模式，再逐步学习其他模式，这样会有事半功倍的效果。

27.3 新的知识新的起点

经过了设计模式的学习，读者就已经可以进行可以复用的面向对象的软件设计了，但是在软件设计中，还有很多其他的模式可以学习，比如企业级模式、架构模式、敏捷模式等。在这里简单地列出进阶模式的大纲，有兴趣的读者可以参照学习。

27.3.1 Java EE设计模式

Java EE就是Java企业版，它是Java程序开发中最重量级的技术，构建大型企业级分布式应用是它的主要应用领域。

Java EE设计模式如表27-2所示。

表27-2 Java EE设计模式

模式分类	模式名称
表现层设计模式	前端控制器
	应用程序控制器
	页面控制器
	上下文对象模式
	拦截过滤器模式
	视图助手模式
	组合视图模式
	分发者视图模式
业务层设计模式	服务到工作者模式
	服务定位器模式
	业务代理模式
	会话外观模式
	应用程序服务模式
	业务接口模式
集成层设计模式	数据访问对象模式
	过程访问对象模式
	服务触发器模式
	Web服务代理模式
横切设计模式	验证和授权实施者模式
	审核拦截器模式
	域服务所有者事务模式

27.3.2 架构风格模式

架构风格就是一组原则。你可以把它看成是一组为系统家族提供抽象框架的粗粒度模式。架构风格能改进分块，还能频繁出现的问题提供解决方案，以此促进设计重用。架构风格模式如表27-3所示。

表27-3 架构风格模式

架构风格	描述
客户端-服务器	将系统分为两个应用，其中客户端向服务器发送服务请求
基于组件的架构	把应用设计分解为可重用的功能、逻辑等组件，这些组件的位置相互透明，只暴露明确定义的通信接口
分层架构	把应用的关注点分割为堆栈组（层）
消息总线	指接收、发送消息的软件系统。消息基于一组已知格式，系统无需知道实际接收者就能互相通信
N层/三层架构	用与分层风格差不多的方式将功能划分为独立的部分，每个部分是一层，分别处于完全独立的计算机上
面向对象	该架构风格是将应用或系统任务分割成单独、可重用、可自给的对象，每个对象包含数据，以及与对象相关的行为
分离表现层	将处理用户界面的逻辑从用户界面（UI）视图和用户操作的数据中分离出来
面向服务架构（SOA）	是指那些利用契约和消息将功能暴露为服务、消费功能服务的应用

27.3.3 架构模式

一个架构模式描述了软件系统里的基本的结构组织或纲要。架构模式提供一些事先定义好的子系统，指定它们的责任，并给出把它们组织在一起的法则和指南，因此它也称做系统模式。

架构模式分为：

- MVC（模型-视图-控制器）模式；
- Layers（分层）模式，有时也称Tiers模式；
- Blackboard（黑板）模式；
- Broker（中介）模式；
- Distributed Process（分散过程）模式；
- Microkernel（微核）模式。

架构模式常常分为如下的几种类型。

- From Mud to Structure型：帮助架构师将系统进行合理划分，避免形成一个对象的海洋（A sea of objects）。包括Layers（分层）模式、Blackboard（黑板）模式、Pipes/Filters（管道/过滤器）模式等。
- 分散系统（Distributed Systems）型：为分散式系统提供完整的架构设计，包括像Broker（中介）模式等。
- 人机互动（Interactive Systems）型：支持包含有人机互动介面的系统的架构设计，如MVC（Model-View-Controller）模式、PAC（Presentation-Abstraction-Control）模式等。
- Adaptable Systems型：支持应用系统适应技术的变化、软件功能需求的变化。如Reflec-

tion（反射）模式、Microkernel（微核）模式等。

27.3.4 分层架构模式

企业应用系统中最常见的架构模式就是分层架构模式。架构设计的一个主要目的，就是把系统划分成为很多“板块”。划分的方式通常有两种，一种是横向的划分，一种是纵向划分。较常见的划分方式是纵向划分，比如一个网站系统可以这样划分：

- 页面层：也就是用户界面，负责显示数据、接收用户输入的信息。
- 业务层：封装了必要的商业逻辑，负责根据商业逻辑决定显示什么数据，以及如何根据用户输入的数据进行计算。
- 数据库：负责存储数据，按照查询要求提供所存储的数据。
- 操作系统层：比如Linux或者Solaris等。
- 硬件层：比如HP-UX服务器等。

分层架构模式的优点是：

- 任何一层的变化都可以很好地控制在这一层，而不会影响到其他各层。
- 更容易容纳新的技术和变化。分层架构模式容许任何一层变更其所使用的技术。

27.4 学习建议

设计模式的学习是一个漫长的过程，学习者必须有一定的编码经验才会对某个模式有更深刻的理解，所以对于学习设计模式我们有以下几点建议：

- 学会使用UML表达思想。
- 在编程中思考重构。
- 思考模式的演化过程。
- 自己创造新的模式。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396；(010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路173信箱

电子工业出版社总编办公室

邮 编：100036

经典 实用 权威

设计模式从入门到精通

本书使用Java语言来描述经典的GoF的23种设计模式，在讲解过程中涉及JDK6.0中的新特性，全书采用案例驱动的形式，以一个完整的超市系统案例统领了全部知识点。

设计模式从入门到精通

Ubuntu Linux 从入门到精通(版本9)

会声会影X2 中文版从入门到精通

UG NX 6中文版从入门到精通

Crystal Reports 2008水晶报表从入门到精通

Windows Server 2008中文版从入门到精通

3ds Max 2010 从入门到精通

AutoCAD 2010中文版从入门到精通

SolidWorks 2010中文版从入门到精通

Pro/Engineer野火5.0中文版从入门到精通

MATLAB 7.6从入门到精通

Oracle 11g从入门到精通

SQL从入门到精通

Mac OS X 10.5中文版从入门到精通

Mac OS X 10.5从入门到精通(中文版)



责任编辑：李红玉
文字编辑：易 昆
封面设计：李 娜



ISBN 978-7-121-11560-8



9 787121 115608 >

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

定价：62.00元